

INDICE

DESARROLLO DE MÓDULOS PARA ODOO.....	2
PASOS PARA CREAR UN MÓDULO	3
FORMA 1- Comando scaffold	3
FORMA2- Manual	3
DESCRIPCIÓN DE LOS FICHEROS	5
• FICHERO __INIT__.PY	5
• FICHERO __MANIFEST__.PY	5
• FICHERO MODELS.PY	6
• FICHERO VIEWS.XML	8
INSTALACIÓN	12
POSIBLES ERRORES.....	13
DEFINIR VISTAS en VIEWS.XML	13
ANEXO1: Tipos de campos y opciones:.....	15
Atributos de campo comunes	16
ANEXO2: Buenas prácticas:.....	17
Bibliografía:	18

NOTA: Descargar Archivo academia.rar para seguir la explicación mejor.

DESARROLLO DE MÓDULOS PARA ODOO

Cuando instalamos un sistema, nos encontramos con situaciones en las que no es posible cubrir todas las necesidades del cliente con los componentes existentes. El sistema es lo suficientemente flexible como para dotarnos de todas las herramientas necesarias para incrementar de forma fácil y eficiente la funcionalidad, adaptándola a cualquier necesidad que nos encontremos. Para desarrollar esta actividad nos podemos centrar en dos procesos de desarrollo:

Variar el código del servidor modificando su funcionamiento desde el código fuente.

Crear pequeños "paquetes de software" de expansión al código existente, Estos paquetes se denominan módulos.

El uso y funcionamiento de los módulos ya lo conocemos. Aquellos módulos que nosotros creemos tendrán un tratamiento similar a cualquier otro, siendo necesario actualizar el sistema desde Administración para detectar el modulo, instalarlo en el sistema de forma general, actualizarlo y borrarlo como los demás y utilizarlo una vez configurado integrándose completamente con el sistema.

Para crear un módulo hay que crear un directorio que contendrá todos los ficheros necesarios y situarlo dentro del directorio 9 del servidor copiándolo o con un fichero comprimido zip. Los ficheros a crear, sus nombres y estructura está determinada por el sistema, no pudiendo cambiar la mayoría de ellos.

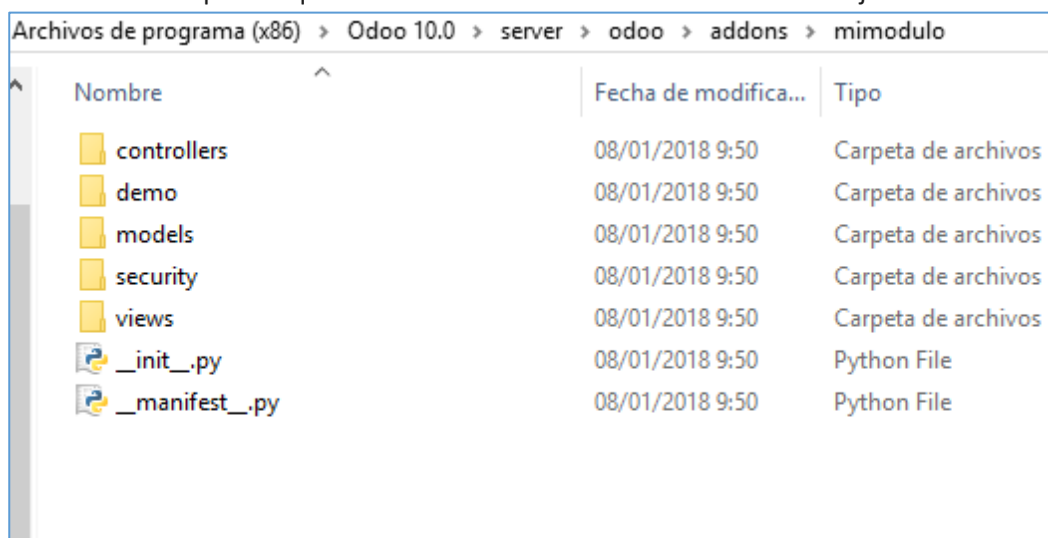
Odoo sigue una arquitectura similar a MVC, y pasaremos por las capas durante nuestra implementación de la aplicación de tareas pendientes:

- La capa del modelo, que define la estructura de los datos de la aplicación
- La capa de vista: La capa de vista describe la interfaz de usuario. Las vistas se definen mediante XML, que es utilizado por el marco de cliente web para generar vistas HTML con datos.
- La capa del controlador, que soporta la lógica de negocio de la aplicación

PASOS PARA CREAR UN MÓDULO

FORMA 1- Comando scaffold

- Crear la estructura de ficheros y directorios del fichero de forma automática:
 - a) Windows: Abrir cmd ejecutado como administrador:
 1. Desplazarse hasta el directorio addons (`cd "C:\Program Files (x86)\Odoo 10.0\server\odoo\addons"`)
 2. Ejecutar el comando odoo:
`"C:\Program Files (x86)\Odoo 10.0\server\odoo-bin" scaffold mimodulo`
 3. Comprobar que se ha creado la estructura del módulo debajo de addons:



The screenshot shows a Windows File Explorer window with the address bar displaying the path: `Archivos de programa (x86) > Odoo 10.0 > server > odoo > addons > mimodulo`. The main pane shows a list of files and folders created by the scaffold command. The table below represents the content of this window.

Nombre	Fecha de modifica...	Tipo
controllers	08/01/2018 9:50	Carpeta de archivos
demo	08/01/2018 9:50	Carpeta de archivos
models	08/01/2018 9:50	Carpeta de archivos
security	08/01/2018 9:50	Carpeta de archivos
views	08/01/2018 9:50	Carpeta de archivos
__init__.py	08/01/2018 9:50	Python File
__manifest__.py	08/01/2018 9:50	Python File

FORMA2- Manual

- Nos desplazamos al directorio **addons** situado dentro de la instalación del Server.
 - Windows: `C:\Program Files (x86)\Odoo 10.0\server\odoo\addons`.
 - Ubuntu `/usr/lib/python2.7/dist-packages/openerp/addons`

Y creamos un nuevo directorio para nuestro modulo: **mimodulo**.

- Añadimos los ficheros obligatorios: **__init__.py** (dos guiones bajos al inicio y final) y **__manifest__.py** vacíos.
- Creamos el fichero de definición del objeto (puede tener otro nombre que incluiremos en **__init__.py**): **models.py** también vacío.
- Añadimos un fichero llamado **templates.xml** (puede tener otro nombre que incluiremos en **__manifest__.py**) que contendrá datos para el sistema (Acciones, vistas, menús, informes)
- Cuando este programado el contenido de todos los ficheros tendremos que abrir una sesión con el servidor, desplazarnos hasta Administración / Módulos y

actualizar la lista de módulos. En el momento que nuestro modulo sea detectado lo podemos instalar, apareciendo en la pantalla principal.

b) Ubuntu: en la terminal

1. Ejecutar: `sudo python /usr/bin/odoo.py scaffold mimodulo /usr/lib/python2.7/dist-packages/opener/addons`

```
mad2@mad2:/usr/lib/python2.7$ sudo python /usr/bin/odoo.py scaffold mimodulo /usr/lib/python2.7/dist-packages/opener/addons/
[sudo] password for mad2:
mad2@mad2:/usr/lib/python2.7$
```

2. Comprobar la creación de la estructura bajo `/usr/lib/python2.7/dist-packages/opener/addons`

```
drwxr-xr-x  3 root root 4096 ene  3 17:41 mimodulo
mad2@mad2:/usr/lib/python2.7/dist-packages/opener/addons$ cd mimodulo
mad2@mad2:/usr/lib/python2.7/dist-packages/opener/addons/mimodulo$ ls
controllers.py  __init__.py  __openerp__.py  templates.xml
demo.xml       models.py    security
mad2@mad2:/usr/lib/python2.7/dist-packages/opener/addons/mimodulo$
```

Donde:

- **scaffold**: comando para crear un la estructura genérica de un módulo en odoo.
- **mimodulo**: Nombre del modulo
- `/usr/lib/python2.7/dist-packages/opener/addons/`: Ruta en donde se creará la estructura del módulo

DESCRIPCIÓN DE LOS FICHEROS

- FICHERO `__init__.py`

Aquí se importa todos los archivos y directorios que contienen código python, éste archivo hace que Odoo reconozca al directorio o capeta *mimodulo* como un módulo.

Es ejecutado al inicio del programa. En él incluiremos los archivos de Python que necesiten ser cargados, generalmente solo es necesaria la carga del fichero principal.

Por lo tanto, si creamos un archivo "modulo.py", que contiene la descripción de nuestros objetos, la incluimos en una línea en `__init__.py`.

Si hemos creado la estructura de forma automática, contendrá lo siguiente:

```
# -*- coding: utf-8 -*-  
  
from . import controllers  
from . import models
```

- FICHERO `__manifest__.py`

Más información en: <https://www.odoo.com/documentation/10.0/reference/module.html>

Es un diccionario en Python para agregar las descripciones del módulo, como autor, versión, etc.

Cualquier módulo que creamos debe contener un archivo con este nombre `__manifest__.py` el cual debe ubicarse en la raíz de nuestro módulo.

Si hemos creado la estructura de forma automática, sólo tendremos que modificar los valores que vienen en él. De otra manera tendremos que escribirlos con la nomenclatura:

'VALOR': "ASIGNACION",

```
# -*- coding: utf-8 -*-  
{  
    'name': "mimodulo",  
  
    'summary': """  
        Short (1 phrase/line) summary of the module's purpose, used as  
        subtitle on modules listing or apps.openerp.com""",  
  
    'description': """  
        Long description of module's purpose  
        """,  
  
    'author': "My Company",  
    'website': "http://www.yourcompany.com",  
  
    # Categories can be used to filter modules in modules listing  
    #  
    # for the full list  
    'category': 'Uncategorized',  
    'version': '0.1',  
  
    # any module necessary for this one to work correctly  
    'depends': ['base'],  
  
    # always loaded  
    'data': [  

```

Check

https://github.com/odoo/odoo/blob/master/odoo/addons/base/module/module_data.xml

```

        # 'security/ir.model.access.csv',
        'views/views.xml',
        'views/templates.xml',
    ],
    # only loaded in demonstration mode
    'demo': [
        'demo/demo.xml',
    ],
}

```

Los valores a usar son:

- **name:** Describe el nombre del módulo, es **obligatorio**.
- **author:** El nombre del autor.
- **version:** Número que indica la versión del módulo, teniendo que ser incrementado en uno al menos en el número menor para que se realice una actualización.
- **description:** Descripción de la función del módulo.
- **website:** Url con la dirección del programador.
- **depends:** Es una lista Python ([valor, valor, ..]) que indica todos los modules que deben estar instalados obligatoriamente para que este funcione. Esta lista debe incluir **obligatoriamente el modulo base** como mínimo.
Ejemplo de varias depedencias: ['base', 'sale', 'mrp', 'mrp_operation',],
- **data:** una lista que incluye los archivos de tus vistas (MVC, las vistas son definiciones de las interfaces), datos, informes, asistentes y similares. Si hay varios archivos de vistas, se escriben separados por comas.
Ejemplo de varias vistas:
['ventas_view.xml', 'produccion_view.xml', 'clases2/clasebasica_view.xml'],
- **category:** Una descripción de la categoría y subcategoría del módulo.
- **demo:** Lista de ficheros XML con los datos de prueba.
- **auto_install:** Determina si el módulo se puede o no instalar. Valores posibles: True o False. Utilizado para módulos linkados a otros.
 - True: el módulo se instalará si todas sus dependencias están instaladas. Es decir, en cuanto sus dependencias estén instaladas se instala automáticamente él sólo.
 - Por defecto: False

● FICHERO MODELS.PY

Es donde definiremos el modelo, el nombre de la clase y cada uno de los atributos que queremos ver en el módulo.

Al generar los ficheros automáticamente el nombre de class y el atributo `_name` (que se asignará a la tabla) serán el mismo.

```
# -*- coding: utf-8 -*-
```

```
from odoo import models, fields, api
```

```
# class mimodulo(models.Model):
#     _name = 'mimodulo.mimodulo'

#     name = fields.Char()
#     value = fields.Integer()
#     description = fields.Text()
#
```

Donde:

- Todas las clases heredan de models.Model
- class: nombre de la clase. Recomendable poner nombremodulo_nombre clase. Este nombre no se utiliza realmente.
- _name: indica el nombre que tendrá el modelo, es decir la tabla en la BD. Recomendabl poner como en la clase_nombre modulo.nombreclase. Este nombre se utilizará a la hora de referenciar los datos.
- name=fields.Char() define la plantilla para la definición de las columnas de la tabla. En este caso la columna se llamará name y será de tipo Char.

Nomenclatura para los campos:

Nombre = fields.Tipo(opcion1=valor1, opcion2=valor2,...)

Tener en cuenta varias cosas:

- Tipo de campo: Empieza por mayúsculas
- NO utilizar mayúsculas ni en el nombre de la tabla (_name) ni en el nombre de los campos.
- Si no hay opciones a definir, se escriben los paréntesis sin nada en medio. Ejemplo: Float()
- Si en vez del nombre del campo queremos que aparezca otro valor, utilizaremos entre los paréntesis la opción string="NUEVA PALABRA". Si no tiene esa opción definida, al visualizar los datos utilizará el nombre del campo.

Otros ejemplos de campos:

- Fecha=fields.Date,
- Descrip=fields.Char(string="Descripcion",size=30,required=True,help="Pequeña descripción de 30 caracteres")
- Edad=fields.Integer(size=10, required=true)
- Nota=fields.Float(string="Nota final", size=209)
- Activo=fields.Boolean(default=True)

Podemos agregar más campos a nuestro modelo, por ejemplo campos de tipo fecha, float, select, boolean, many2one, etc.

Ejemplo de opciones posibles de los campos:

```
name = fields.Char(
    string="Name",           # Etiqueta opcional para el Campo
    compute="_compute_name_custom", # Transforma un campo normal en campo función
    store=True,              # Si el campo es calculado, esta propiedad lo grabara en base
                             # de datos como una columna.
    select=True,             # Fuerza el índice en un campo.
    readonly=True,           # Campo solo lectura en las Vistas.
    inverse="_write_name"    # Recalcular cuando ocurre un evento
    required=True,           # Campo Obligatorio.
    translate=True,          # Convertir texto para múltiples traducciones.
```

```

help='Blabla',          # Globo de ayuda.
company_dependent=True,  # Convierte el campo a campo de tipo propiedad(property)
search='_search_function' # Función de Búsqueda para campos calculados.
copy=False # !! Evita copiar el valor de este campo cuando duplicamos un registro.
)

```

Más detalles en:

<http://odoo-new-api-guide-line.readthedocs.io/en/latest/fields.html>

<https://www.odoo.com/documentation/10.0/reference/orm.html#fields>

<http://poncesoft.blogspot.com.es/2016/01/nuevas-propiedades-para-los-campos-en.html>

EJEMPLO:

```
# -*- coding: utf-8 -*-
```

```
from odoo import models, fields, api
```

```

class academia_curso(models.Model):
    _name = 'academia.curso'
    name = fields.Char(string="Titulo", required=True)
    descripcion = fields.Text(string="Descripcion")

```

• FICHERO VIEWS.XML

Creando Acciones y Menús

Las acciones y los menús son registros en la base de datos que se declaran en un archivo xml, las acciones pueden ser activados de las formas siguientes:

- Haciendo clic en los items del menú
- Haciendo clic en botones definidos en la vista del modelo.

Este fichero tiene 2 etiquetas y dentro de ellas escribiré la configuración que considere.

```

<odoo>
  <data>

  </data>
</odoo>

```

ELEMENTOS A DEFINIR EN ESTE FICHERO Y FORMA DE HACERLO:

1. ACCION – Obligatorio

<https://www.odoo.com/documentation/10.0/reference/actions.html>

Con la etiqueta <record model="ir.actions.act_window" id="XXXX"> definimos una acción de ventana para abrir una vista.

Esta acción se une a los menús a través de la propiedad action del menú que se rellena con el valor de la propiedad id definida en la acción.

Se define el objeto que necesitamos abrir a través de los atributos name y res_model.

Una acción en Odoo se define mediante las etiquetas

```
<record>  
  
</record>
```

La etiqueta **<record>** tiene los atributos:

- **model** : Indica el modelo de acciones de ventana ir.actions.act_window

```
<record model="ir.actions.act_window"
```

- **id**: Identificador único para cada acción

```
<record model="ir.actions.act_window" id="XXXXX">
```

- Dentro del record se definen **fields name=** que indicarán diferentes cosas:

1. `<field name="name">NOMBREACCION</field>`

Permite asignar un nombre a la acción. Obligatorio

2. `<field name="res_model">miclase</field>`

Asocia el modelo con la acción. Obligatorio

3. `<field name="view_type">form</field>`

Tipo de vista por defecto: form / tree. Recordad los tipos de vistas y sus modos:

Tipos de vista	Modos de vista permitidos
Tree (Árbol)	<ul style="list-style-type: none">• tree (lista)• form (formulario)
Form (Formulario)	<ul style="list-style-type: none">• tree (lista)• form (formulario)• kanban (imágenes)• graph (gráfico)• gantt (diagrama de gantt)• calendar (calendario)• search (búsqueda)• inheritance (herencia)

4. `<field name="view_mode">tree,form</field>`

Dentro de ese tipo de vista, qué modos se van a definir

5. `<field name="help" type="html">`
`<p class="oe_view_nocontent_create">SOY TU AYUDA. HAZME CASO POR`
`FAVOR</p>`
`</field>`

Texto de ayuda en el caso de que no haya ningún registro en el modelo. Opcional

EJEMPLO ACCION:

`<!-- Definimos la accion -->`

```
<record model="ir.actions.act_window" id="course_list_action">
  <field name="name">Cursos</field>
  <field name="res_model">academia.curso</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form</field>
  <field name="help" type="html">
    <p class="oe_view_nocontent_create">Cree el primer curso</p>
  </field>
</record>
```

6. MENÚ-Obligatorio

Un menú en Odoo se define mediante las etiquetas

`<menuitem atributos... />`

La etiqueta `<menuitem>` tiene los atributos:

7. **id**: identificador único para cada menú. *Obligatorio*
8. **name**: Nombre del menú (etiqueta). *Obligatorio*
9. **parent**: Indica si un menú es hijo de otro menú
10. **action**: Permite asociar un menú a una acción.
11. **sequence**: Indica la secuencia, en qué prioridad u orden irá el menú.

EJEMPLO MENÚS ENLAZADOS

`<!-- menu superior o principal: no hereda de otro menu -->`

```
<menuitem id="main_academia"

  name="Cursos"

  sequence="10" />
```

```

        <!-- Definimos un menu de solo vista, este menu tiene como menu padre a
        Cursos. En odoo es el de color morado-->

```

```

<menuitem id="cursos_menu_1"

        name="Cursos academicos"

        parent="main_academia"

        />

```

```

        <!-- Definimos otro menu que tiene como menu padre a Curso academicos. A
        este le asignamos la acción-->

```

```

<menuitem id="cursos_menu2"

        name="Cursos"

        parent="cursos_menu_1"

        action="listar_cursos"

        sequence="10" />

```

EJEMPLO ACCION y MENÚ juntos en el FICHERO TEMPLATES.XML

```

<?xml version="1.0" encoding="UTF-8"?>
<odoo>
    <data>
        <!-- Definimos la acción -->

        <record model="ir.actions.act_window" id="listar_cursos ">
            <field name="name">Cursos</field>
            <field name="res_model">curso</field>
            <field name="view_type">form</field>
            <field name="view_mode">tree,form</field>
            <field name="help" type="html">
                <p class="oe_view_nocontent_create">Cree el primer curso</p>
            </field>
        </record>

        <!-- menu superior o principal: no hereda de otro menu -->

        <menuitem id="main_academia"

                name="Cursos"

                sequence="10" />

```

```
<!-- Definimos un menu de solo vista, este menu tiene como menu padre a
Cursos. En odoo es el de color morado-->
```

```
<menuitem id="cursos_menu_1"

    name="Cursos academicos"

    parent="main_academia"

    sequence="10" />
```

```
<!-- Definimos otro menu que tiene como menu padre a Curso academicos. A
este le asignamos la acción-->
```

```
<menuitem id="cursos_menu2"

    name="Cursos" parent="cursos_menu_1"

    action="listar_cursos"

    sequence="10" />

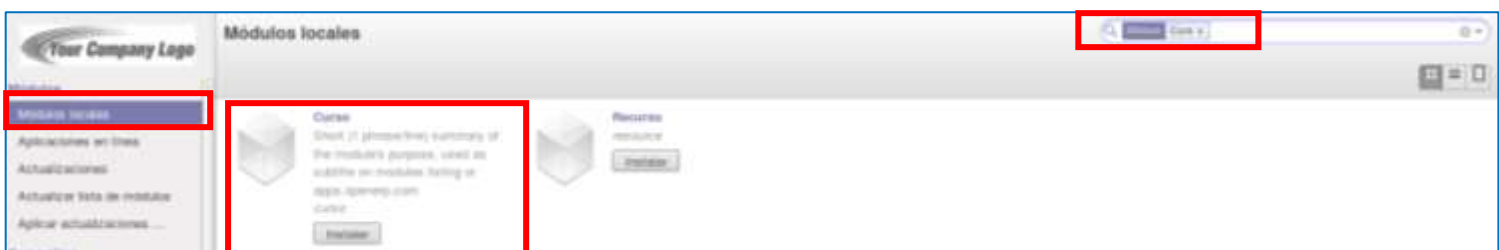
</data>
</odoo>
```

INSTALACIÓN

1. Una vez configurados los archivos, en la interfaz gráfica actualizar la lista de módulos.



2. Buscar el módulo entre los Módulos locales



3. Instalarlo



POSIBLES ERRORES

1. El módulo se ha instalado pero tras desinstalarlo no desaparece, hay un nuevo módulo que no ve, se ha quedado algún menú en la barra de menús de un módulo desinstalado,.... → Reiniciar el servicio de odoo.
2. Se han realizado cambios en el modelo, pero al reinstalarlo esos cambios no se ven. → Mirar dentro de la carpeta models y eliminar los ficheros y compilados (extensión pyc) Tras esto, instalar de nuevo.

EJERCICIO PRIMER MODULO

DEFINIR VISTAS en VIEWS.XML

Las vistas se definen como elementos `<record>` `</record>`

Llevarán como atributos:

- **model**: Indica el modelo vista

```
<record model="ir.ui.view">
```
- **id**: Identificador único de la vista. Seguir nomenclatura: `clase_tipovista_view` para evitar duplicaciones

```
<record model="ir.ui.view" id="clase_tree_view">
```

- Dentro del record se definen **fields** que indicarán diferentes cosas:

1. Para asignar un nombre a la vista

```
<field name="name">miclase.tree</field>
```

2. Para definir el modelo de donde saldrá los datos

```
<field name="model">miclase</field>
```

3. Para definir los campos que componen la vista, en este caso concretamente la vista tipo Tree

```
<field name="type">tree</field>
```

```
<field name="arch" type="xml">
```

```
    <tree string="Titulo de la vista">
        <field name="campo1 del modelo"/>
        <field name=" campo2"/>
        <field name=" campo3"/>
        <field name="campo4"/>
    </tree>
</field>
```

4. Para definir la vista Form y los campos que componen la vista.

```
<field name="name">miclase.form</field>
```

```
<field name="type">form</field>
```

```
<field name="arch" type="xml">
```

```
    <form string="Titulo de la vista">
        <field name="campo1 del modelo"/>
        <field name=" campo2"/>
        <field name=" campo3"/>
        <field name="campo4"/>
    </form>
</field>
```

EJEMPLO DE LOS RECORDS DE VISTAS

```
<record model="ir.ui.view" id="curso_tree_view">
    <field name="name">curso.tree</field>
    <field name="model">curso</field>
    <field name="type">tree</field>
    <field name="arch" type="xml">
        <tree string="CURSOS">
            <field name="name"/>
            <field name="description"/>
        </tree>
    </field>
</record>
```

```
<record model="ir.ui.view" id="curso_form_view">
    <field name="name">curso.form</field>
    <field name="model">curso</field>
    <field name="type">form</field>
```

```

        <field name="arch" type="xml">
            <form string="CURSO">
                <field name="name"/>
                <field name="description"/>
            </form>
        </field>
</record>

```

ANEXO1: Tipos de campos y opciones:

Estos son los argumentos de posición estándar esperados por cada uno de los tipos de campo:

- Char espera un segundo tamaño de argumento opcional para el tamaño máximo de texto. Se recomienda no usarlo a menos que exista un requisito de negocio que lo requiera, como un número de seguro social con una longitud fija.

```
name = fields.Char('Name', 40)
```

```
name = fields.Char(string='Name', size=40)
```

- Texto difiere de Char, ya que puede albergar contenido de texto multilínea, pero espera los mismos argumentos.

```
desc = fields.Text('Description')
```

- Selection es una lista de selección desplegable. El primer argumento es la lista de opciones seleccionables y el segundo es el título de la cadena. El elemento de selección es una lista de tuplas ('value', 'Título'), para el valor almacenado en la base de datos y la correspondiente descripción de interfaz de usuario. Cuando se extiende a través de la herencia, el argumento selection_add está disponible para añadir nuevos elementos a una lista de selección existente.

```
state = fields.Selection(
    [('draft', 'New'), ('open', 'Started'),
     ('done', 'Closed')], 'State')
```

- Html se almacena como un campo de texto, pero tiene un manejo específico en la interfaz de usuario, para la presentación de contenido HTML. Por razones de seguridad, se desinfectan de forma predeterminada, pero este comportamiento se puede sobrescribir.

```
docs = fields.Html('Documentation')
```

- Integer sólo espera un argumento de cadena para el título del campo.

```
sequence = fields.Integer('Sequence')
```

- Float tiene un segundo argumento opcional, una tupla (x,y) con la precisión del campo: x es el número total de dígitos; De éstos, y son dígitos decimales.

```
perc_complete = fields.Float('% Complete', (3, 2))
```

- Los campos Date y Datetime sólo esperan la cadena de texto como un argumento de posicional. Por razones históricas, el ORM maneja sus valores en un formato de cadena. Las funciones auxiliares se deben utilizar para convertirlas en objetos de fecha real. También los valores de fecha y hora se almacenan en la base de datos en tiempo UTC pero presentadas en hora local, utilizando las preferencias de zona horaria del usuario.

```
date_effective = fields.Date('Effective Date')
date_changed = fields.Datetime('Last Changed')
```

- Boolean tiene valores True o False y sólo tiene un argumento de posición para la cadena de texto.

```
fold = fields.Boolean('Folded?')
```

- Binary almacena datos binarios de tipo archivo y también espera sólo el argumento de cadena. Pueden ser manejados por código Python usando cadenas codificadas en base64.

```
image = fields.Binary('Image')
```

Atributos de campo comunes

Los campos tienen atributos que se pueden establecer al definirlos. Dependiendo del tipo de campo, algunos atributos pueden ser pasados en posición, sin una palabra clave de argumento, como se muestra en la sección anterior.

Por ejemplo, `name=fields.Char('Name', 40)` podría hacer uso de argumentos posicionales. Usando los argumentos de la palabra clave, lo mismo se podría escribir como `name=fields.Char (size=40, string='Name')`.

Más información sobre argumentos de palabras clave:

<https://docs.python.org/2/tutorial/controlflow.html#keyword-arguments>.

Todos los atributos disponibles se pueden pasar como un argumento de palabra clave. Estos son los atributos generalmente disponibles y las palabras clave de argumento correspondientes:

- `string` es la etiqueta por defecto del campo, que se utilizará en la interfaz de usuario. Excepto para los campos de selección y relacionales, es el primer argumento posicional, por lo que la mayoría de las veces no se utiliza como argumento de palabra clave.
- `default` establece un valor predeterminado para el campo. Puede ser un valor estático, como una cadena o una referencia callable, ya sea una función con nombre o una función anónima (una expresión lambda).
- `size` sólo se aplica a los campos Char y puede establecer un tamaño máximo permitido. La mejor práctica actual es no usarla a menos que sea realmente necesaria.
- `translate` se aplica sólo a los campos Char, Text y Html, y hace que el contenido del campo se pueda traducir, manteniendo valores diferentes para diferentes idiomas.
- `help` proporciona el texto para las sugerencias que se muestran a los usuarios.
- `readonly=True` hace que el campo por defecto no sea editable por la interfaz de usuario. Esto no se aplica a nivel API; Es sólo una configuración de interfaz de usuario.

- `required=True` hace obligatorio el campo por defecto en la interfaz de usuario. Esto se aplica en el nivel de base de datos mediante la adición de una restricción NOT NULL en la columna.
- `index=True` creará un índice de base de datos en el campo.
- `copy=False` tiene el campo ignorado cuando se utiliza la función de registro duplicado, método ORM `copy()`. Los campos no relacionales son `copyable` de forma predeterminada.
- `groups` permite limitar el acceso y la visibilidad del campo a sólo algunos grupos. Espera una lista separada por comas de IDs XML para grupos de seguridad, como `groups='base.group_user, base.group_system'`.
- `states` espera un diccionario que asigna valores para los atributos UI que dependen de los valores del campo `state`. Por ejemplo: `states = {'done': [('readonly', True)]}`. Los atributos que se pueden utilizar son `readonly`, `required` e `invisible`.

ANEXO2: Buenas prácticas:

Estructura del módulo

- Carpeta `models` para los archivos `.py`
- Carpeta `views` para las vistas
- Carpeta `reports` para los informes
- Carpeta `wizard` para los asistentes (vistas y código)
- Carpeta `security` para seguridad.
- Otras carpetas son obligatorias: `i18n`, `static`
- Un archivo para cada modelo.

Campos

- Nombres y etiquetas en inglés
- Descriptivos, pero cortos
- Utilizar related, calculados, almacenados de forma adecuada
- ... e inteligentemente

Código

- Hacer imports relativos
- Nombres de clases CamelCase
- Nombres de métodos en minúsculas y con _ como unión
- No sobrescribir métodos completos
- Escoger la técnica de sobrescribir antes o después de la acción
- PEP8

BIBLIOGRAFIA:

<https://www.odoo.com/documentation/10.0/reference/module.html>
<https://github.com/EmperoVE/odoo-10-development-essential-es/blob/master/SUMMARY.md>
https://es.slideshare.net/pedromanuelbaezaromero/2015-0615-jornadas-odoo-buenas-prcticas?next_slideshow=1