

Capítulo 7. Conjuntos de registros: trabajar con datos del modelo

En los capítulos anteriores, presentamos una descripción general de la creación de modelos y cómo cargar y exportar datos de los modelos. Ahora que tenemos nuestro modelo de datos y algunos datos para trabajar, es hora de aprender más sobre cómo podemos interactuar programáticamente con ellos.

El **mapeo relacional de objetos (ORM)** que respalda nuestros modelos proporciona algunos métodos para esta interacción, llamada la **interfaz de programación de aplicaciones (API)**. Estos comienzan con las **operaciones** básicas de **Crear, Leer, Actualizar, Eliminar (CRUD)**, pero también incluyen otras operaciones, como la exportación de datos e importar, o funciones de utilidad para ayudar a la interfaz de usuario y la experiencia. También proporciona algunos decoradores que ya vimos en los capítulos anteriores. Estos nos permiten, al agregar nuevos métodos, informar al ORM cómo deben manejarse.

En este capítulo, cubriremos los siguientes temas:

- Usar el comando de shell para explorar interactivamente la API ORM
- Comprender el entorno y el contexto de ejecución
- Consulta de datos utilizando conjuntos de registros y dominios
- Acceso a datos en conjuntos de registros
- Escribir en registros
- Composición de conjuntos de registros
- Uso de transacciones de bases de datos y SQL de bajo nivel

◀ Sección anterior Sección (/book/business/9781789532470/6/ch06lvl1sec69/summary)

siguiente ➤ (/book/business/9781789532470/7/ch07lvl1sec70/technical-requirements)





Requerimientos técnicos

Los ejemplos de código de este capítulo se ejecutarán en un shell interactivo y no requieren ningún código de los capítulos anteriores.

◀ Sección anterior Sección (/book/business/9781789532470/7)

siguiente ▶ (/book/business/9781789532470/7/ch07lvl1sec71/using-the-shell-command)



Usando el comando de shell

Python tiene una interfaz de línea de comandosesa es una excelente manera de explorar su sintaxis. Del mismo modo, Odoo también tiene una característica equivalente, donde podemos probar comandos interactivamente para ver cómo funcionan. Este es el `shell` comando.

Para usarlo, ejecute Odoo con el `shell` comando y la base de datos que se utilizará, como se muestra aquí:

Dupdo

```
$ ./odoo-bin shell -d 12-library
```

Debería ver la secuencia de inicio del servidor habitual en el terminal hasta que se detenga en un `>>>` indicador de Python esperando su entrada.



Nota

Changed in Odoo 9 La función de shell se agregó en la versión 9.0. Para la versión 8.0, hay un módulo de comunidad con puertos para agregarlo. Una vez descargado e incluido en la ruta de los complementos, no es necesaria ninguna instalación adicional. Se puede descargar desde <https://www.odoo.com/apps/modules/8.0/shell/> (<https://www.odoo.com/apps/modules/8.0/shell/>).

Aquí, `self` representará el registropara el `Administrator` usuario, como puede confirmar escribiendo lo siguiente:

Dupdo

```
>>> self
res.users(1,)
>>> self._name
'res.users'
>>> self.name
'OdooBot'
>>> self.login
'__system__'
```

En la sesión de shell aquí, inspeccionamos nuestro entorno:

- El `self` comando representa un `res.users` conjunto de registros que contiene solo el registro con la `1` ID.
- El nombre del modelo de conjunto de registros, inspección `self._name` , es `'res.users'` , como se esperaba.
- El valor para el campo de **nombre de** registro es `OdooBot` .
- El valor para el campo de **inicio de sesión de** registro es `__system__` .



Nota

Changed in Odoo 12 El usuario con superusuario ID 1 ha cambiado de administrador al usuario del sistema interno que no tiene su propio inicio de sesión. El administrador ahora es el usuario con ID 2 y no es un superusuario, pero por defecto las aplicaciones lo incluyen en todos los grupos de seguridad. La razón principal de esto es evitar que los usuarios realicen actividades cotidianas con la cuenta de superusuario. Hacerlo es peligroso porque omite todas las reglas de acceso y permite mezclar datos inconsistentes, como las relaciones entre empresas. Ahora está destinado a ser utilizado solo para la resolución de problemas o operaciones muy específicas entre empresas.

Al igual que con Python, puede salir del símbolo utilizando **Ctrl + D**. Esto también cerrará el proceso del servidor y lo devuelve al indicador de shell del sistema.

◀

Sección anterior Sección (/book/business/9781789532470/7/ch07lvl1sec70/technical-requirements)

siguiente ▶ (/book/business/9781789532470/7/ch07lvl1sec72/the-execution-environment)



El entorno de ejecución.

El shell del servidor proporciona una `self` referencia similar a lo que harías encontrar dentro de un método del `res.users` modelo de usuario.

Como hemos visto, `self` es un conjunto de registros. **Recordsets** llevan con la información del entorno, incluido el usuario que navega por los datos y la información de contexto adicional, como el idioma y la zona horaria.

En las siguientes secciones, aprenderemos sobre los atributos disponibles en el entorno de ejecución, la utilidad del contexto del entorno y cómo modificar este contexto.

Atributos del entorno

Podemos inspeccionar la corriente entorno con el siguiente código:

Dupdo

```
>>> self.env
<openrp.api.Environment object at 0xb3f4f52c>
```

El entorno de ejecución en `self.env` tiene los siguientes atributos disponibles:

- `env.cr` es el cursor de la base de datos que se está utilizando.
- `env.user` es el registro para el usuario actual.
- `env.uid` es la ID para el usuario de la sesión. Es lo mismo que `env.user.id`.
- `env.context` es un diccionario inmutable con un contexto de sesión.

El entorno también proporciona acceso al registro donde están disponibles todos los modelos instalados. Por ejemplo, `self.env['res.partner']` devuelve una referencia al modelo de pareja. Luego podemos usar `search()` o `browse()` en él para recuperar conjuntos de registros:

Dupdo

```
>>> self.env['res.partner'].search([('name', 'like', 'Ad')])
res.partner(10, 35, 3)
```

En este ejemplo, el conjunto de registros devuelto para el `res.partner` modelo contiene tres registros, con los ID 10, 35 y 3. El conjunto de registros no está ordenado por ID, porque se utilizó el orden predeterminado para el modelo correspondiente. En el caso del modelo de socio, el valor predeterminado `_order` es `display_name`.

El contexto del entorno.

El **contexto** es el diccionario que lleva datos de sesión que se puede usar tanto en la interfaz de usuario del lado del cliente como en el ORM del lado del servidor y la lógica empresarial.

En el lado del cliente, puede llevar información de una vista a la siguiente, como el ID del registro activo en la vista anterior, después de seguir un enlace o un botón, o puede proporcionar valores predeterminados para usar en la siguiente vista.

En el lado del servidor, algunos valores de campo de conjunto de registros pueden depender de la configuración regional proporcionada por el contexto. En particular, la `lang` clave afecta el valor de los campos traducibles. El contexto también puede proporcionar señales para el código del lado del servidor.

Por ejemplo, la `active_test` tecla, cuando se establece en `False`, cambia el comportamiento del `search()` método del ORM, ignorando la `active` marca en los registros, de modo que los registros inactivos (borrados por software) también se devuelven.

El contexto inicial del cliente web se ve así:

Dupdo

```
{'lang': 'en_US', 'tz': 'Europe/Brussels', 'uid': 2}
```

Puede ver la `lang` clave con el idioma del usuario `tz`, la información de la zona horaria y `uid` con el ID de usuario actual.

El contenido de los registros puede ser diferente según el contexto actual:

- `translated` los campos pueden tener valores diferentes según el `lang` idioma activo
- `datetime` los campos pueden mostrar diferentes horas dependiendo de la `tz` zona horaria activa

Al abrir un formulario desde un enlace o un botón en una vista anterior, `active_id` se agrega una clave al contexto, con el ID del registro en el que estábamos ubicados, en el formulario de origen. En el caso particular de las vistas de lista, tenemos una `active_ids` clave de contexto que contiene una lista de las ID de registro seleccionadas en la lista anterior.

En el lado del cliente, el contexto se puede usar para establecer valores predeterminadosvalores o active filtros predeterminados en la vista de destino, utilizando teclas con los prefijos `default_` o `default_search_`.

Aquí hay unos ejemplos:

- Para establecer el usuario actual como un valor predeterminado del `user_id` campo, utilizamos `{'default_user_id': uid}`
- Para tener un `filter_my_books` filtro activado por defecto en la vista de destino, utilizamos `{'default_search_filter_my_tasks': 1}`

Modificar el entorno de ejecución del conjunto de registros

El entorno de ejecución del conjunto de registros es inmutable, por lo que no se puede modificar. Pero nosotros podemos crear un entorno modificado y luego ejecutar acciones usándolo.

Para hacerlo, podemos hacer uso de los siguientes métodos:

- `env.sudo(user)` se proporciona con un registro de usuario y devuelve un entorno con ese usuario. Si no se proporciona ningún usuario, `__system__` se utilizará el superusuario raíz, lo que permite ejecutar operaciones específicas, sin pasar por las reglas de seguridad.
- `env.with_context(<dictionary>)` reemplaza el contexto por uno nuevo.
- `env.with_context(key=value, ...)` modifica el contexto actual, estableciendo valores para algunas de sus claves.

Además, tenemos la `env.ref()` función, tomar una cadena con un identificador externo y devolver un registro, como se muestra aquí:

Dupdo

```
>>> self.env.ref('base.user_root')
s.users(1,)
```

◀ Sección anterior Sección (/book/business/9781789532470/7/ch07lvl1sec71/using-the-shell-command)

siguiente ▶ (/book/business/9781789532470/7/ch07lvl1sec73/querying-data-with-recordsets-and-domains)



Consulta de datos con conjuntos de registros y dominios

Dentro de un método o una conchasesión, `self` representa el actualmodelo, y solo podemos acceder a los registros de ese modelo. Para acceder a otrosmodelos, debemos usar `self.env` . Por ejemplo, `self.env['res.partner']` devuelve una referenciaal modelo Partner (que también es un vacío conjunto de registros).

Luego podemos usar `search()` o `browse()` en él para recuperar conjuntos de registros, y los `search()` métodos usan una expresión de dominio para definir los criterios de selección de registros.

Crear conjuntos de registros

El `search()` método toma una expresión de dominio.y devuelve un conjunto de registros con los registros que coinciden con esas condiciones. Un dominio vacío `[]` devolverá todos los registros.



Nota

Si el modelo tiene el `active` campo especial, por defecto solo se considerarán los registros con `active=True` .

También se pueden usar los siguientes argumentos de palabras clave:

- `order` es una cadena que se utilizará como `ORDER BY` cláusula en la consulta de la base de datos. Esta suele ser una lista de nombres de campo separados por comas. Cada nombre de campo puede ir seguido de la `DESC` palabra clave, para indicar un orden descendente.
- `limit` establece un número máximo de registros para recuperar.
- `offset` ignora los primeros `n` resultados; se puede usar `limit` para consultar bloques de registros a la vez.

A veces, solo necesitamos saber la cantidad de registros que cumplen ciertas condiciones. Para eso podemos usar `search_count()` , que devuelve el recuento de registros en lugar de un conjunto de registros. Ahorra el costo de recuperar una lista de registros solo para contarlos, por lo que es mucho más eficiente cuando todavía no tenemos un conjunto de registros y solo queremos contar la cantidad de registros.

El `browse()` método toma una lista de ID o una ID única y devuelve un conjunto de registros con esos registros. Esto puede ser conveniente en los casos en que ya conocemos los ID de los registros que queremos.

Aquí se muestran algunos ejemplos de uso:

Dupdo

```
>>> self.env['res.partner'].search([('name', 'like', 'Ag')])
res.partner(9, 31)
>>> self.env['res.partner'].browse([9, 31])
res.partner(9, 31)
```

Expresiones de dominio

El **dominio** se usa para filtrar registros de datos Usan una sintaxis específica que el ORM de Odoo analiza para producir las WHERE expresiones SQL que consultarán la base de datos.

Una expresión de dominio es una lista de condiciones. Cada condición es una ('<field_name>', '<operator>', <value>') tupla. Por ejemplo, esta es una expresión de dominio válido, con una sola condición:

```
[('is_done', '=', False)]
```

La siguiente es una explicación de cada uno de estos elementos:

- ▶ `<field_name>` es el campo que se está filtrando y puede usar notación de puntos para campos en modelos relacionados.
- ▶ `<value>` se evalúa como una expresión de Python. Puede usar valores literales, como números, booleanos, cadenas o listas, y puede usar campos e identificadores disponibles en el contexto de evaluación. En realidad, hay dos posibles contextos de evaluación para dominios:
 - ▶ Cuando se usa en el lado del cliente, como en acciones de ventana o atributos de campo, los valores de campo sin procesar utilizados para representar la vista actual están disponibles, pero no podemos usar la notación de punto en ellos
 - ▶ Cuando se usa en el lado del servidor, como en las reglas de registro de seguridad y en el código Python del servidor, la notación de puntos se puede usar en los campos, ya que el registro actual es un objeto
- ▶ El `<operator>` puede ser uno de los siguientes:
 - ▶ Los operadores de comparación son habituales `<` , `>` , `<=` , `>=` , `=` , y `!=` .
 - ▶ `'=like'` y `'=ilike'` coincide con un patrón, donde el símbolo de subrayado `_` , coincide con cualquier carácter individual y el símbolo de porcentaje `%` , coincide con cualquier secuencia de caracteres.
 - ▶ `'like'` coincide con un `'%value%'` patrón. `'ilike'` es similar pero no distingue mayúsculas de minúsculas. Los operadores `'not like'` y `'not ilike'` también están disponibles.
 - ▶ `'child of'` encuentra los valores secundarios en una relación jerárquica para los modelos configurados para admitir relaciones jerárquicas.
 - ▶ `'in'` y `'not in'` se usan para verificar la inclusión en una lista dada, por lo que el valor debe ser una lista de valores. Cuando se usa en un campo de relación de **muchos**, el `in` operador se comporta como un `contains` operador.
 - ▶ `'not in'` es lo opuesto `in` y se usa para verificar que el valor no esté contenido en una lista de valores.

Una expresión de dominio es una lista de elementos y puede contener varias tuplas de condición. Por defecto, estas condiciones se combinarán implícitamente utilizando el operador lógico AND. Esto significa que solo devolverá registros que cumplan con todas las condiciones.

También se pueden usar operadores lógicos explícitos: el símbolo de ampersand `'&'` , para operaciones AND (el valor predeterminado) y el símbolo de tubería `'|'` , para operaciones OR. Estos voluntadoperar en los siguientes dos elementos, trabajando de forma recursiva. Veremos esto con más detalle en un momento.

Nota

Para una definición un poco más formal, una expresión de dominio usa notación de prefijo, también conocida como notación polaca, donde los operadores preceden a los operandos. Los operadores AND y OR son operadores binarios, mientras que NOT es un operador unario.

El signo de exclamación `'!'` , representa el operador NOT y está disponible y opera en el siguiente elemento. Por lo tanto, debe colocarse antes del elemento que se va a negar. Por ejemplo, la `['!', ('is_done', '=', True)]` expresión filtrará todos los registros no hechos.

El **siguiente elemento** también puede ser un elemento de operador que actúa sobre sus siguientes elementos, definiendo condiciones anidadas. Un ejemplo puede ayudarnos a comprender esto mejor.

En las reglas de registro del lado del servidor, podemos encontrar expresiones de dominio similares a esta:

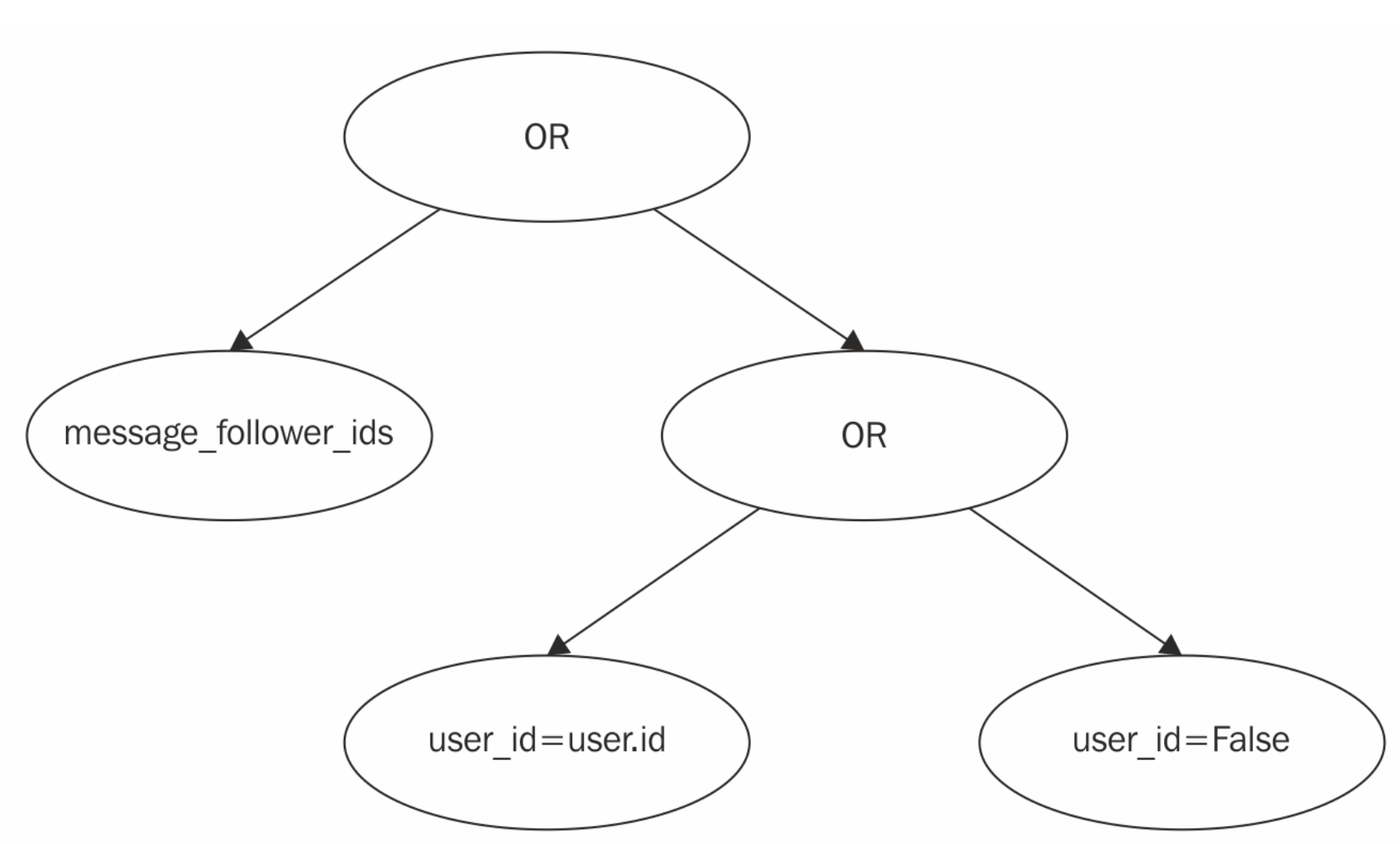
Dupdo

```
[ '|',
  ('message_follower_ids', 'in', [user.partner_id.id]),
  '|',
  ('user_id', '=', user.id),
  ('user_id', '=', False)
]
```

Este dominio filtra todos los registros donde el usuario actual está en la lista de seguidores y es el usuario responsable, o no tiene un conjunto de usuarios responsables.

El primer operador `'|'` **OR** actúa según la condición del seguidor más el resultado de la siguiente condición. La siguiente condición es nuevamente la unión de otras dos condiciones: registros en los que el ID de usuario es el usuario de la sesión actual o no está establecido.

El siguiente diagrama ilustra la representación de árbol de sintaxis abstracta de la expresión de dominio de ejemplo anterior:



Acceso a datos en conjuntos de registros

Una vez que tenemos un conjunto de registros, queremos para inspeccionar los datos que contiene. En el siguiente En las secciones, exploraremos cómo acceder a los datos en conjuntos de registros.

Podemos obtener valores de campo para registros individuales, llamados singletons. Los campos relacionales tienen propiedades especiales, y podemos usar la notación de puntos para navegar a través de registros vinculados. Finalmente, tenemos algunas consideraciones sobre cómo manejar registros de fecha y hora y convertirlos entre diferentes formatos.

Acceso a datos en registros

El caso especial de un conjunto de registros con un solo registro se llama **singleton**. Los solteros todavía son un conjunto de registros y se puede usar donde se espera un conjunto de registros.

Pero a diferencia de los conjuntos de registros de elementos múltiples, los singletons pueden acceder a sus campos utilizando la notación de puntos, de esta manera:

Dupdo


```
>>> print(self.name)
OdooBot
```

En el siguiente ejemplo, podemos ver que el mismo `self` conjunto de registros singleton también se comporta como un conjunto de registros, y podemos iterarlo. Tiene solo un registro, por lo que solo se imprime un nombre:

Dupdo

```
>>> for rec in self:
    print(rec.name)
OdooBot
```

Intentar acceder a valores de campo en conjuntos de registros con más de un registro dará como resultado un error, por lo que esto puede ser un problema en los casos en los que no estamos seguros de si estamos trabajando con un conjunto de registros singleton. Para los métodos diseñados para funcionar solo con un singleton, podemos verificar esto usando `self.ensure_one()` al principio. Levantará un error si `self` no es un singleton.

 Nota

Un registro vacío también es un singleton. Esto es conveniente ya que acceder a los valores de campo devolverá un `None` valor, en lugar de generar un error. Esto también es cierto para los campos relacionales, y el acceso a registros relacionados mediante la notación de puntos no generará errores.

Accediendo a campos relacionales

Como vimos anteriormente, los modelos pueden tener campos relacionales: **muchos a uno**, **uno a muchos** y **muchos a muchos**. Estos tipos de campo tienen conjuntos de registros como valores

En el caso de muchos a uno, el valor puede ser un singleton o un conjunto de registros vacío. En ambos casos, podemos acceder directamente a sus valores de campo. Como ejemplo, las siguientes instrucciones son correctas y seguras:

Dupdo

```
record = self.env['model.name'].search([('id', 1)])
record.name
```

```
>>> self.company_id
res.company(1,)
>>> self.company_id.name
'YourCompany'
>>> self.company_id.currency_id
res.currency(1,)
>>> self.company_id.currency_id.name
'EUR'
```

Convenientemente, un conjunto de registros vacío también se comporta como singleton, y el acceso a sus campos no devuelve un error, sino que solo regresa `False`. Debido a esto, podemos atravesar registros utilizando la notación de puntos sin preocuparnos por errores de valores vacíos, como se muestra aquí:

Dupdo

```
>>> self.company_id.parent_id
res.company()
>>> self.company_id.parent_id.name
False
```

Acceso a valores de fecha y hora

En conjuntos de registros, `date` y los `datetime` valores son representado como Python nativo objetos. Por ejemplo, cuando buscamos la última `admin` fecha de inicio de sesión:

Dupdo

```
>>> self.browse(2).login_date
datetime.datetime(2018, 11, 2, 16, 47, 57, 327756)
```

Desde la fecha y hora los valores son objetos de Python, tienen toda la manipulación características disponibles para estos objetos.



Nota

Changed in Odoo 12 Los valores de campo `date` y `datetime` se representan como objetos de Python, a diferencia de las versiones anteriores de Odoo, donde `date` y los `datetime` valores se representan como cadenas de texto. Estos valores de tipo de campo aún se pueden establecer utilizando representaciones de texto, como en versiones anteriores de Odoo.

Las fechas y horas se almacenan en la base de datos en un ingenuo formato de **hora universal coordinada (UTC)**, no consciente de la zona horaria. Los `datetime` valores que se ven en los conjuntos de registros también están en UTC. Cuando se presentan al usuario en el cliente web, los `datetime` valores se convierten en la zona horaria del usuario, de acuerdo con la configuración de la zona horaria de la sesión actual, almacenada en la `tz` clave de contexto, como `{'tz': 'Europe/Brussels'}`. Esta conversión es responsabilidad del cliente web, ya que no la realiza el servidor.

Por ejemplo, una fecha y hora de las 12:00 a.m. ingresada por un usuario de Bruselas (UTC + 1) se almacena en la base de datos a las 10:00 a.m. UTC, y un usuario de Nueva York (UTC-4) la verá a las 06:00 a.m. .



Nota

Las marcas de tiempo del mensaje de registro del servidor Odoo usan la hora UTC y no la hora del servidor local.

La conversi3n opuesta, desde la zona horaria de la sesi3n a UTC, tambi3n debe ser realizada por el cliente web, al enviar la `datetime` entrada del usuario al servidor.

Los objetos de fecha se pueden comparar y se pueden restar para encontrar el tiempo transcurrido entre ambas fechas. Este tiempo transcurrido es un `timedelta` objeto. Y los `timedelta` objetos se pueden sumar o restar `date` y `datetime` objetos, realizando `date` operaciones aritm3ticas.

Estos objetos son proporcionados por el `datetime` m3dulo de biblioteca est3ndar de Python . Aqu3 hay una muestra de las operaciones esenciales que podemos hacer con 3l:

Dupdo

```
>>> from datetime import date
>>> date.today()
datetime.date(2018, 11, 3)
>>> from datetime import timedelta
>>> timedelta(days=7)
datetime.timedelta(7)
>>> date.today() + timedelta(days=7)
datetime.date(2018, 11, 10)
```

Una referencia completa para el `date` , `datetime` y `timedelta` tipos de datos se puede encontrar en: <https://docs.python.org/3.6/library/datetime.html> (<https://docs.python.org/3.6/library/datetime.html>) .

Odoo tambi3n proporciona algunas funciones de conveniencia adicionales en el `odoo.tools.date_utils` m3dulo. Estas funciones son las siguientes:

- `start_of(value, granularity)` es el inicio de un per3odo de tiempo especificado con el granularity- `year` , `quarter` , `month` , `week` , `day` ,o `hour` .
- `end_of(value, granularity)` es el final de un per3odo de tiempo para la granularidad dada.
- `add(value, **kwargs)` agrega un intervalo de tiempo al valor dado. Los `**kwargs` argumentos deben ser utilizados por un `relativedelta` objeto para definir el intervalo de tiempo. Estos argumentos pueden ser `years` , `months` , `weeks` , `days` , `hours` , `minutes` ,y as3 sucesivamente.
- `subtract(value, **kwargs)` resta un intervalo de tiempo al valor dado.

El `relativedelta` objeto es de la `dateutil` biblioteca y puede realizar `date` operaciones aritm3ticas usando `months` o `years` (la biblioteca `timedelta` est3ndar de Python solo apoyos `days`). La documentaci3n para ello se puede encontrar en: <https://dateutil.readthedocs.io>. (<https://dateutil.readthedocs.io>)

Lo siguienteson algunos ejemplos del uso de los anteriores funciones:

Dupdo

```
>>> from odoo.tools import date_utils
>>> from datetime import datetime
>>> date_utils.start_of(datetime.now(), 'week')
datetime.datetime(2018, 10, 29, 0, 0)
>>> date_utils.end_of(datetime.now(), 'week')
datetime.datetime(2018, 11, 4, 23, 59, 59, 999999)
>>> from datetime import date
>>> date_utils.add(date.today(), months=2)
datetime.date(2019, 1, 3)
>>> date_utils.subtract(date.today(), months=2)
datetime.date(2018, 9, 3)
```



Estas funciones de utilidad también están expuestas tanto en `odoo.fields.Date` los `odoo.fields.Datetime` objetos como en los objetos, junto con estos:

- `fields.Date.today()` devuelve una cadena con la fecha actual en el formato esperado por el servidor y utilizando UTC como referencia. Esto es adecuado para calcular los valores predeterminados. En este caso, deberíamos usar solo el nombre de la función sin paréntesis.
- `fields.Datetime.now()` devuelve una cadena con la corriente `datetime` en el formato esperado por el servidor utilizando UTC como referencia. Esto es adecuado para calcular los valores predeterminados.
- `fields.Date.context_today(record, timestamp=None)` devuelve una cadena con el actual `date` en el contexto de la sesión. El valor de zona horaria se toma del contexto del registro. El `timestamp` parámetro opcional es un `datetime` objeto y se usará en lugar de la hora actual, si se proporciona.
- `fields.Datetime.context_timestamp(record, timestamp)` convierte un `datetime` valor ingenuo (sin zona horaria) en una zona horaria consciente `datetime` . La zona horaria se extrae del contexto del registro, de ahí el nombre de la función.

Convertir fechas y horas representadas por texto

Antes de Odoo 12, necesitábamos para convertir representaciones de texto por `date` y `datetimes` antes podríamos hacer cualquier cálculo con ellos. Se proporcionan algunas herramientas para ayudar con esta conversión de texto a tipos de datos nativos y luego a texto.

Estos son útiles cuando trabajando con versiones anteriores de Odoo y todavía relevante para Odoo 12; para este caso, necesitamos trabajar con una fecha que nos llegue formateada como texto.

Para facilitar la conversión entre formatos, ambos `fields.Date` y los `fields.Datetime` objetos proporcionan estas funciones:

- `to_date` Convierte una cadena en un `date` objeto.
- `to_datetime(value)` Convierte una cadena en un `datetime` objeto.
- `to_string(value)` convierte un objeto `date` u `datetime` en una cadena en el formato esperado por el servidor Odoo, hasta la versión 11.

Los formatos de texto utilizados por ambas funciones son los valores predeterminados predefinidos de Odoo, definidos a continuación:

- `odoo.tools.DEFAULT_SERVER_DATE_FORMAT`
- `odoo.tools.DEFAULT_SERVER_DATETIME_FORMAT`

Se asignan a `%Y-%m-%d` y `%Y-%m-%d %H:%M:%S` , respectivamente.

Un ejemplode un usode `from_string` es el siguiente:

Dupdo

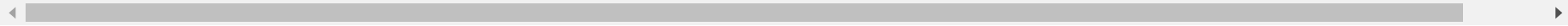
```
>>> from odoo import fields
>>> fields.Datetime.to_datetime('2018-11-21 23:11:55')
datetime.datetime(2018, 11, 21, 23, 11, 55)
```

Para otra fechay formatos de tiempo, el `strptime` método deel `datetime` objeto se puede usar:

Dupdo

```
>>> from datetime import datetime
>>> datetime.strptime('03/11/2018', '%d/%m/%Y')
datetime.datetime(2018, 11, 3, 0, 0)
```

◀ Sección anterior Sección (/book/business/9781789532470/7/ch07lvl1sec73/querying-data-with-recordsets-and-dc
siguiente ▶ (/book/business/9781789532470/7/ch07lvl1sec75/writing-on-records)



Escribir en registros

Tenemos dos formas diferentes de escribir en registros: mediante la asignación directa de estilo de objeto y el `write()` método. El primero es más simple de usar, pero solo funciona con un registro a la vez y puede ser menos eficiente. Dado que cada asignación realiza una `write` operación, pueden ocurrir cálculos redundantes. Este último requiere una sintaxis especial para escribir en campos de relación, pero un solo comando puede escribir en varios campos y registros, y recálculos de campo más eficientes.

Escribir con asignación de valor de estilo de objeto

Los conjuntos de registros implementan el registro activo modelo. Esto significa que podemos asignar valores para ellos, y estos cambios se harán persistentes en la base de datos. Esta es una forma intuitiva y conveniente de manipular datos, pero solo funciona con un campo y un registro a la vez.

Aquí hay un ejemplo:

Dupdo

```
>>> root = self.env['res.users'].browse(1)
>>> print(root.name)
OdooBot
>>> root.name = 'Superuser'
>>> print(root.name)
Superuser
```

Mientras se usa el patrón de registro activo, el valor de los campos relacionales se puede establecer asignando un conjunto de registros.

Para los campos muchos a uno, el valor asignado debe ser un registro único (un conjunto de registros singleton).

Para muchos campos, el valor también se puede asignar con un conjunto de registros, reemplazando la lista de registros vinculados, si los hay, con uno nuevo. Aquí, se permite un conjunto de registros con cualquier tamaño.

Escribir con el método write ()

También podemos usar el `write()` método para actualizar varios campos de varios registros al mismo tiempo, usando una sola instrucción de base de datos. Por lo tanto, su uso debe preferirse en casos donde la eficiencia es importante.

El `write()` método toma un diccionario para asignar campos a valores. Estos se actualizan en todos los elementos del conjunto de registros y no se devuelve nada, como se muestra aquí:

Dupdo

```
>>> recs = Partner.search( [('name', 'ilike', 'Azure')] )
>>> recs.write({'comment': 'Hello!'})
True
```

A diferencia de la asignación de estilo de objeto, cuando usamos el `write()` método, no podemos asignar objetos de conjunto de registros a campos relacionales directamente. En su lugar, debemos usar las ID de registro que deben extraerse del conjunto de registros.

Al escribir en un campo de varios a uno, el valor a escribir debe ser la ID del registro relacionado. Por ejemplo, en lugar de , deberíamos usar `self.write({'user_id': self.env.user})` `self.write({'user_id': self.env.user.id})`

Al escribir en un campo de muchos, el valor a escribir debe ser la misma sintaxis especial utilizada en los archivos de datos XML, que se describe en el Capítulo 5 (/book/business/9781789532470/5) , **Importar, Exportar y Datos del módulo** .

Por ejemplo, establezcamos una lista de autores de libros en `author1` y `author2` , que son dos registros de socios. El `|` operador de tubería puede unir registros para crear un conjunto de registros, por lo que, mediante la asignación de estilo de objeto, podríamos escribir lo siguiente:

Dupdo

```
publisher.child_ids = author1 | author2
```

Usando el `write()` método, la misma operación se ve así:

Dupdo

```
book.write( { 'child_ids': [(6, 0, [author1.id, author2.id])] } )
```

Recordando la escriturasintaxis explicada en el Capítulo 5 (/book/business/9781789532470/5) , **Importar, Exportar y Datos del módulo** , los más comunes los comandos son los siguientes:

- `(4, id, _)` para agregar un registro
- `(6, _, [ids])` para reemplazar la lista de registros vinculados con la lista proporcionada

Escribir valores de fecha y hora

Desde Odoo 12, fecha y horalos campos se pueden escribir utilizando valores en Python nativetipos de datos, tanto mediante asignación directa como mediante `write()` .

Todavía podemos escribir la fecha y la hora usando valores representados por texto:

Dupdo

```
>> demo = self.search([('login', '=', 'demo')])
>>> demo.login_date
False
>>> demo.login_date = '2018-01-01 09:00:00'
>>> demo.login_date
datetime.datetime(2018, 1, 1, 9, 0)
```

Crear y eliminar registros

El `write()` método se usa para escribirfechas en registros existentes. Pero nosotros tambiennecesita crear y eliminar registros. Esto se hace usando los métodos `create()` y `unlink()` modelo.

El `create()` método toma un diccionario con los campos y valores para el registro que se creará, utilizando la misma sintaxis que `write()` . Los valores predeterminados se aplican automáticamente como se esperaba, que se muestra aquí:

Dupdo

```
>>> Partner = self.env['res.partner']
>>> new = Partner.create({'name': 'ACME', 'is_company': True})
>>> print(new)
res.partner(59,)
>>> print(new.customer) # customer flag is True by default
True
```

El `unlink()` método elimina los registros en el conjunto de registros, como se muestra aquí:

Dupdo

```
>>> rec = Partner.search([('name', '=', 'ACME')])
>>> rec.unlink()
2018-11-02 17:01:01,433 18380 INFO 12-library odoo.models.unlink: User #1 deleted mail.message records with IDs: [27]
2018-11-02 17:01:01,458 18380 INFO 12-library odoo.models.unlink: User #1 deleted ir.attachment records with IDs: [385,
384, 383]
2018-11-02 17:01:01,466 18380 INFO 12-library odoo.models.unlink: User #1 deleted res.partner records with IDs: [46]
2018-11-02 17:01:01,474 18380 INFO 12-library odoo.models.unlink: User #1 deleted mail.followers records with IDs: [6]
True
```

Anteriormente, vimos varios mensajes de registro para otros registros que se eliminaban. Estas son cascadas de eliminación de registros relacionados con la eliminación del socio.

También tenemos disponible el `copy()` método modelo, útil para duplicarUn registro existente. Acepta un argumento opcional con los valores a cambiar.en el nuevo registro, por ejemplo, para crear un `new` usuario copiando del `demo` usuario:

Dupdo

```
>>> demo = self.env.ref('base.user_demo')
>>> new = demo.copy({'name': 'Daniel', 'login': 'daniel', 'email':''})
```

Los campos con el `copy=False` atributo no se copiarán automáticamente. Demasiados campos relacionales tienen este indicador deshabilitado de forma predeterminada y, por lo tanto, no se copiarán.



Composición de conjuntos de registros

Los conjuntos de registros admiten algunos adicionalesoperaciones Podemos verificar si un registro está incluido o no en un conjunto de registros. Si `x` es un conjunto de registros singleton y `my_recordset` es un conjunto de registros que contiene muchos registros, podemos usar lo siguiente:

- `x in my_recordset`
- `x not in my_recordset`

Las siguientes operaciones también están disponibles:

- `recordset.ids` devuelve la lista con los ID de los elementos del conjunto de registros.
- `recordset.ensure_one()` comprueba si es un registro único (singleton); si no es así, se `ValueError` genera una excepción.
- `recordset.filtered(func)` devuelve un conjunto de registros filtrado y `func` puede ser una función o una expresión separada por puntos que representa una ruta de campos a seguir, como se muestra en los ejemplos que siguen.
- `recordset.mapped(func)` devuelve una lista de valores asignados. En lugar de una función, se puede usar una cadena de texto para asignar el nombre del campo.
- `recordset.sorted(func)` devuelve un conjunto de registros ordenado. En lugar de una función, se puede usar una cadena de texto para ordenar el nombre del campo. Un `reverse=True` argumento opcional también está disponible.

Aquí hay algunos ejemplos de uso para estas funciones:

Dupdo

```
>>> rs0 = self.env['res.partner'].search([])
>>> len(rs0)# how many records?
40>>> starts_A = lambda r: r.name.startswith('A')
>>> rs1 = rs0.filtered(starts_A)
>>> print(rs1)
res.partner(41, 14, 35)>>> rs1.sorted(key=lambda r: r.id, reverse=True)
res.partner(41, 35, 14)
>>> rs2 = rs1.filtered('is_company')
>>> print(rs2)
res.partner(14,)>>> rs2.mapped('name')
['Azure Interior']>>> rs2.mapped(lambda r: (r.id, r.name))
[(14, 'Azure Interior')]
```

Seguramente querremos agregar, eliminar o reemplazar los elementos en estos campos relacionados, y esto lleva a la pregunta: ¿cómo se pueden manipular los conjuntos de registros?

Los conjuntos de registros son inmutables, lo que significa que sus valoresNo se puede modificar directamente. En cambio, modificar un conjunto de registros significa componer un nuevo conjunto de registros basado en los existentes.

Una forma de hacerlo es utilizando las operaciones de conjunto compatibles:

- `rs1 | rs2` es la operación de conjunto de **unión** y da como resultado un conjunto de registros con todos los elementos de ambos conjuntos de registros.
- `rs1 + rs2` es la operación de conjunto de **suma** para concatenar ambos conjuntos de registros en uno. Puede resultar en un conjunto con registros duplicados.

- `rs1 & rs2` es la operación del conjunto de **intersección** y da como resultado un conjunto de registros con solo los elementos presentes en ambos conjuntos de registros.
- `rs1 - rs2` es la operación de conjunto de **diferencias** y da como resultado un conjunto de registros con los `rs1` elementos no presentes en `rs2`.

La notación de corte también se puede usar, como se muestra en estos ejemplos:

- `rs[0]` y `rs[-1]` recuperar el primer elemento y el último elemento, respectivamente.
- `rs[1:]` da como resultado una copia del conjunto de registros sin el primer elemento. Esto produce los mismos registros `rs - rs[0]` pero conserva su orden.



Nota

Changed in Odoo 10 Desde Odoo 10, la manipulación de conjuntos de registros ha preservado el orden. Esto es diferente a las versiones anteriores de Odoo, donde no se garantizaba que la manipulación de conjuntos de registros conservara el orden, aunque se sabía que la adición y el corte en rebanadas mantenían el orden de los registros.

Podemos usar estas operaciones para cambiar un conjunto de registros eliminando o agregando elementos. Aquí hay unos ejemplos:

- `self.author_ids |= author1` agrega el `author1` registro, si no está en el conjunto de registros
- `self.author_ids -= author1` elimina el registro específico `author1` si está presente en el conjunto de registros
- `self.author_ids = self.author_ids[:-1]` elimina el último registro

Los campos relacionales contienen valores de conjunto de registros. Campos muchos a uno puede contener un conjunto de registros singleton, y muchos campos contienen conjuntos de registros con cualquier número de registros.

◀ Sección anterior Sección (/book/business/9781789532470/7/ch07lvl1sec75/writing-on-records)

siguiente ▶ (/book/business/9781789532470/7/ch07lvl1sec77/low-level-sql-and-database-transactions)



Transacciones de base de datos y SQL de bajo nivel

Se ejecutan operaciones de escritura de base de datos en el contexto de una base de datos transacción. Por lo general, no tenemos que preocuparnos por esto, ya que el servidor se encarga de eso mientras ejecuta métodos modelo.

Pero, en algunos casos, es posible que necesitemos un control más preciso sobre la transacción. Esto se puede hacer a través del `self.env.cr` cursor de la base de datos, como se muestra aquí:

- `self.env.cr.commit()` confirma las operaciones almacenadas en la memoria intermedia de la transacción `write`.
- `self.env.cr.rollback()` cancela las operaciones de la transacción `write` desde el último compromiso o todo si no se realizó ningún compromiso.



Nota

En una sesión de shell, su manipulación de datos no se hará efectiva en la base de datos hasta que la use `self.env.cr.commit()`.

Con el `execute()` método del cursor, podemos ejecutar SQL directamente en la base de datos. Se necesita una cadena con la instrucción SQL para ejecutarse y un segundo argumento opcional con una tupla o una lista de valores para usar como parámetros para el SQL. Estos valores se utilizarán donde `%s` se encuentren los marcadores de posición.



Nota

Caution! Con `cr.execute()` no debemos componer directamente los parámetros de concatenación de consultas SQL. Hacerlo es un riesgo de seguridad bien conocido que puede explotarse mediante ataques de inyección SQL. Utilice siempre los `%s` marcadores de posición y el segundo parámetro para pasar valores.

Si está utilizando una `SELECT` consulta, los registros deben recuperarse. La `fetchall()` función recupera todas las filas como una lista de `tuples`, y las `dictfetchall()` recupera como una lista de diccionarios, como se muestra en el siguiente ejemplo:

Dupdo

```
>>> self.env.cr.execute("SELECT id, login FROM res_users WHERE login=%s OR id=%s", ('demo', 1))
>>> self.env.cr.fetchall()
[(6, 'demo'), (1, '__system__')]
```



También es posible ejecutar instrucciones del **lenguaje de manipulación de datos (DML)**, comocomo `UPDATE` y `INSERT` . Dado que el servidor mantiene cachés de datos, pueden volverse inconsistentes con los datos reales en la base de datos. Por eso, cuando se usa DML sin procesar, los cachés deberíanser despejado después porutilizando `self.env.cache.invalidate()` .



Nota

Caution! Ejecutar SQL directamente en la base de datos puede generar datos inconsistentes. Debe usarlo solo si está seguro de lo que está haciendo.

◀ Sección anterior Sección (/book/business/9781789532470/7/ch07lvl1sec76/composing-recordsets)

siguiente ▶ (/book/business/9781789532470/7/ch07lvl1sec78/summary)



Resumen

En este capítulo, aprendimos cómo trabajar con datos del modelo y realizar operaciones CRUD: crear, leer, actualizar y eliminar datos. Esta es la base para implementar nuestra lógica y automatización de negocios.

Para experimentar con la API ORM, utilizamos el shell interactivo Odoo. Nuestros comandos se ejecutan en un entorno accesible a través de `self.env`. El entorno proporciona acceso al registro del modelo y tiene un contexto que puede proporcionar información relevante para los comandos ejecutados, como el idioma actual `lang` y la zona horaria `tz`.

Conjuntos de registros se crean utilizando los `search(<domain>)` o `browse([<ids>])` ORM métodos. Luego se pueden iterar para acceder a cada singleton (un registro individual). Luego, podemos usar la notación de puntos tipo objeto en singletons para obtener y establecer valores en los registros.

Además de la asignación de valor directo en singletons, también podemos usar `write(<dict>)` para actualizar todos los elementos en un conjunto de registros con un solo comando. Las `create(<dict>)`, `copy(<dict>)` y los `unlink()` comandos se utilizan para crear, duplicar y eliminar registros.

Los conjuntos de registros pueden ser inspeccionados y manipulados. Los operadores de inspección incluyen `in` y `not in`. Los operadores de composición incluyen `|` para unión, y `&` para intersección y corte. Transformaciones disponibles incluyen `.ids`, para extraer la lista de ID, `.mapped(<field>)`, `.filtered(<func>)`, o `.sorted(<func>)`.

Finalmente, las operaciones SQL de bajo nivel y el control de transacciones son posibles a través del `cursor` objeto expuesto en `self.env.cr`.

En el próximo capítulo, agregaremos la capa de lógica de negocios para nuestros modelos, implementando métodos de modelo que usan la API ORM para automatizar acciones.

