Preguntas sobre Odoo

1. ¿Qué es un modelo en Odoo y cómo se define en Python?

Un modelo: Los modelos determinan la estructura lógica de una base de datos y cómo se almacenan, organizan y manipulan los datos. En otras palabras, un modelo es una tabla de información que se puede vincular con otras tablas.

En Python se define un modelo de Odoo de la siguiente forma:

Se importan las dependencias necesarias para crear el modelo en Odoo, se pueden añadir **fields** para los campos, **api** para los decoradores, etc

```
from odoo import models

class ModelName(models.Model):

    _name = “model.name”

    #Atributos de la clase y métodos
```

2. ¿Cómo se define un campo relacionado en un modelo de Odoo? ¿Para qué sirve?

Para definir un campo relacionado en un modelo de Odoo se usa el atributo "related", ya que permite saber del objeto los datos específicos y agregarcelo al nuevo campo. Es decir, se tiene un modelo A y uno B, se crea un campo en el modelo B que es de tipo A, y a otro campo del modelo B se le relaciona un valor del campo de tipo A.

Ejemplo de campo relacionado:

```
from odoo import fields, models

class A(models.Model):

    _name = ‘model.nameA’

    name = fields.Char(string=”Nombre”)

    age = fields.Integer(string=”Edad”)

class B(models.Model):

     _name = ‘model.nameB’

    model_name_a_id = fields.Many2one(‘model.nameA’, string=”Model name 1”)

    age_model_name_a = fields.Integer(string=“Edad”, related=”
model_name_a_id.age”)
```

Estos campos sirven para mostrar información relevante.

Además de los campos con atributos related están los campos que relacionan modelos entre si como los Many2one, One2many y Many2many Los cuales se usan en dependencia de que relación guarden entre ellos.

El campo **Many2one** establece una relación donde múltiples registros de un modelo están relacionados con un solo registro de otro modelo. Es decir, muchos registros apuntan a uno solo.

El campo **One2many** es la contraparte del Many2one. Define una relación donde un registro está relacionado con múltiples registros de otro modelo. Es decir, uno tiene muchos.

El campo **Many2many** establece una relación donde múltiples registros de un modelo pueden estar relacionados con múltiples registros de otro modelo. Es decir, muchos a muchos.

3. ¿Qué son los registros XML en Odoo y en qué casos se utilizan?

Los registros XML (eXtensible Markup Language) son fragmentos de código que describen los datos y la estructura de los elementos en la base de datos de Odoo. Se utilizan para declarar vistas, acciones, menús, secuencias y otros elementos del Sistema.

4. Explica la diferencia entre los métodos @api.model, @api.depends y @api.onchange en los modelos de Odoo.

@api.model

Este decorador se utiliza para definir métodos a nivel del modelo. No está vinculado a un registro específico y no depende de campos específicos. Se usa cuando el método que deseas definir es genérico y no depende de los datos de un registro en particular. Por ejemplo, se puede utilizar para crear o buscar registros.

@api.depends

Este decorador se utiliza para marcar los métodos que dependen de uno o más campos. Se usa para métodos que calculan valores de campos computados. Cuando los campos especificados cambian, el método decorado con @api.depends se ejecuta automáticamente para actualizar el valor del campo computado.

@api.onchange

Este decorador se utiliza para definir métodos que se ejecutan automáticamente cuando cambia el valor de uno o más campos en la vista del formulario. Se usa para actualizar otros campos en función de los cambios realizados en el formulario. No guarda los cambios en la base de datos, solo afecta la vista del formulario.

Preguntas sobre JavaScript

1. Explica la diferencia entre let, const y var en JavaScript.

El **let, const y var** son formas de declarar variables, pero tienen diferencias en cuanto a su comportamiento y alcance.

Las variables declaradas con **var** tienen alcance de función o global si se declaran fuera de una función. Esto significa que son accesibles en cualquier parte de la función donde se declaran. Es posible redeclarar una variable usando var dentro del mismo contexto sin generar un error.

Las variables declaradas con **let** tienen alcance de bloque, lo que significa que solo son accesibles dentro del bloque {} donde se declaran. No es posible redeclarar una variable usando let dentro del mismo bloque.

Las variables declaradas con **const** también tienen alcance de bloque, igual que let. Las variables declaradas con const deben ser inicializadas en el momento de su declaración y no pueden ser reasignadas. Sin embargo, los objetos y arrays declarados con const pueden modificar sus propiedades y elementos.

2. ¿Cuál es la diferencia entre == y === en JavaScript?

== (Comparación abstracta)

Compara dos valores para verificar si son iguales en valor, sin considerar el tipo de datos. Si los valores comparados son de diferentes tipos, JavaScript intentará convertirlos a un tipo común antes de realizar la comparación.

=== (Comparación estricta)

Compara dos valores para verificar si son iguales en valor y tipo de datos. No realiza conversión de tipos. Si los valores comparados son de diferentes tipos, se consideran no iguales.

Preguntas sobre TypeScript

1. ¿Qué ventajas ofrece TypeScript sobre JavaScript?

Las ventajas que posee TypeScript sobre Javascript son las siguientes:

- Sistema de tipos estáticos.
- Detección temprana de errores (ya que muchos problemas que surgirían en tiempo de ejecución en JavaScript se identifican durante la compilación en TypeScript).
- Mejora enormemente la productividad y la experiencia de desarrollo.
- Autocompletado inteligente
- Navegación de código mejorada
- Verificación de errores en tiempo real
- Introduce características avanzadas como interfaces, genéricos y decoradores, que no están disponibles nativamente en JavaScript.
- Al compilar tu código TypeScript a JavaScript estándar, puedes aprovechar las últimas características del lenguaje mientras garantizas la compatibilidad con diferentes entornos y navegadores.

2. Explica la diferencia entre interface y type en TypeScript.

Una **interface** es una forma de definir la estructura de un objeto, especificando los nombres y tipos de sus propiedades. Las interfaces se utilizan principalmente para definir la forma de objetos y funciones. No pueden representar tipos primitivos o uniones. Pueden ser extendidas utilizando la palabra clave **extends** o redeclarando la interfaz con el mismo nombre. No pueden definir uniones directamente.

El **type** se utiliza para crear **alias de tipos**. Los tipos pueden representar cualquier tipo válido de TypeScript, incluidos tipos primitivos, uniones, intersecciones, tuplas y más. Pueden combinarse usando intersecciones (&), pero no se pueden redeclarar. No pueden definir uniones y tipos complejos.

3. ¿Qué es el unknown type en TypeScript y en qué se diferencia de any?

En TypeScript, el tipo unknown es un tipo seguro que representa un valor cuyo tipo puede ser cualquier cosa, pero a diferencia de any, impone restricciones sobre cómo puedes manejar ese valor.

Característica	any	unknown
Asignación	Acepta cualquier tipo de valor	Acepta cualquier tipo de valor
Comprobación de tipos	No requiere comprobación antes de usar	Requiere comprobación antes de usar
Seguridad	Baja, propenso a errores en tiempo de ejecución	Alta, previene errores mediante verificaciones
Acceso a propiedades	Permite acceso sin restricciones	Restringe acceso sin comprobación de tipo
Uso recomendado	Evitar su uso excesivo, último recurso	Preferido cuando el tipo no es conocido de antemano

4. ¿Cómo se define y utiliza un Generic en TypeScript?

Para definir una función genérica, agregas un **parámetro de tipo** entre < y > después del nombre de la función:

```
function identidad<T>(valor: T): T {
    return valor;
}
```

<T>: Declara un parámetro de tipo llamado T.

(valor: T): T: La función recibe un parámetro valor de tipo T y devuelve un valor del mismo tipo T.

Los genéricos son una característica poderosa de TypeScript que te permiten escribir código más flexible y reutilizable. Puedes crear funciones, clases e interfaces que funcionen con cualquier tipo, manteniendo siempre la seguridad de tipos.