

C.F.G.S. DESARROLLO DE APLICACIONES MULTIPLATAFORMA

MÓDULO:

Sistemas de gestión empresarial
(SGEM)

Unidad 5

Desarrollo de componentes para
un sistema ERP-CRM

INDICE DE CONTENIDOS

1.	TÉCNICAS Y ESTÁNDARES: MODELO-VISTA-CONTROLADOR (MVC).....	3
1.1.	INTRODUCCIÓN.....	3
1.2.	ESPECIFICACIONES TÉCNICAS PARA EL DESARROLLO DE COMPONENTES	4
1.3.	ESPECIFICACIONES FUNCIONALES PARA EL DESARROLLO DE COMPONENTES.....	4
1.3.1.	CREAR UN MÓDULO	5
2.	TÉCNICAS DE OPTIMIZACIÓN DE CONSULTAS Y ACCESO A GRANDES VOLÚMENES DE INFORMACIÓN	8
2.1.	OPERACIONES DE CONSULTA: HERRAMIENTAS	9
2.2.	SISTEMAS BATCH INPUTS. GENERACIÓN DE PROGRAMAS DE EXTRACCIÓN PROCESAM. DE DATOS	14
3.	LENGUAJE PROPORCIONADO POR LOS SISTEMAS ERP-CRM	15
3.1.	CARACTERÍSTICAS E INTRODUCCIÓN AL LENGUAJE	15
3.1.1.	INSTALAR PYTHON	16
3.1.1.1.	En Linux.....	16
3.1.1.2.	En Windows XP.....	16
3.1.2.	HACER UNA PRUEBA en modo comando	16
3.1.3.	HACER UNA PRUEBA en el intérprete de Python.....	17
3.1.4.	SALIR.....	17
3.1.5.	MÁS COSAS.....	17
3.2.	DECLARACIÓN DE DATOS	17
3.2.1.	TIPOS DE DATOS BÁSICOS	17
3.2.2.	OPERADORES.....	19
3.3.	ESTRUCTURAS DE PROGRAMACIÓN: COLECCIONES	23
3.4.	SENTENCIAS DEL LENGUAJE.....	26
3.5.	LLAMADAS A FUNCIONES.....	29
3.6.	CLASES Y OBJETOS	31
3.6.1.	DEFINIR LA CLASE	32
3.6.2.	CREAR UN OBJETO de la clase	32
3.6.3.	ACCEDER LOS ATRIBUTOS Y MÉTODOS del objeto	33
3.7.	MÓDULOS Y PAQUETES	33
3.7.1.	SENTENCIA IMPORT	34
3.7.2.	DIFERENCIA ENTRE MÓDULOS Y PAQUETES.....	35
3.8.	LIBRERIAS DE FUNCIONES (APIs).....	35
3.8.1.	SENTENCIA IMPORT	36
3.9.	INSERCIÓN, MODIFICACIÓN Y ELIMINACIÓN DE DATOS EN OBJETOS (APIs).....	38
3.9.1.	1ª FORMA DE CREAR UN MÓDULO EN OPENERP	38
3.9.2.	2ª FORMA DE CREAR UN MÓDULO EN OPENERP	44
4.	ENTORNOS DE DESARROLLO Y HERRAMIENTAS DE DESARROLLO EN SISTEMAS ERP-CRM.....	44
4.1.	DEPURACIÓN DE UN PROGRAMA	45
4.2.	MANEJO DE ERRORES.....	45
5.	FORMULARIOS E INFORMES EN SISTEMAS ERP-CRM.....	46
5.1.	ARQUITECTURA DE FORMULARIOS E INFORMES: ELEMENTOS	47
5.2.	HERRAMIENTAS PARA LA CREACIÓN DE FORMULARIOS E INFORMES.....	48
	TAREAS.....	50

1. TÉCNICAS Y ESTÁNDARES: MODELO-VISTA-CONTROLADOR (MVC)

1.1. INTRODUCCIÓN

¿Utilizas habitualmente hojas de cálculo, como por ejemplo Calc de OpenOffice.org? En ese caso, sabrás que al introducir datos en las celdas, los mismos datos podemos verlos de varias formas. Es posible ver los datos en la hoja de cálculo donde los hemos introducido, o mediante un gráfico que puede ser de barras, circular, etc.

Esos programas implementan el patrón Modelo-Vista-Controlador (MVC), el modelo lo constituyen los datos que hemos introducido en las celdas. Las vistas se encargan de mostrar los datos de la forma que se seleccione.

Patrón: Esquema o estructura de diseño con la que construir sistemas de software

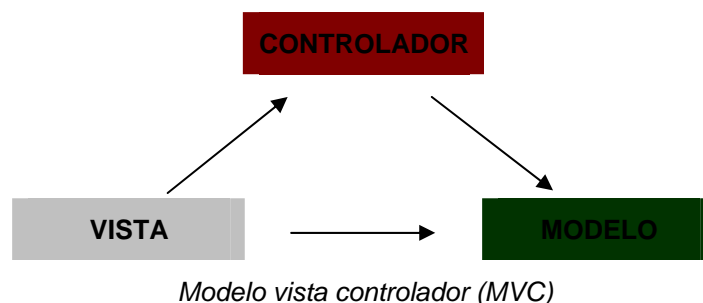
El MVC divide una aplicación en tres componentes:

- Los datos de la aplicación (**modelo**).
- La interfaz del usuario (**vista**).
- El **controlador**, el cual define la forma en que la interfaz reacciona a la entrada del usuario.

Con esto se consigue separar los datos (modelo) de la interfaz de usuario (vista), de manera que los cambios en la interfaz no afectan a los datos y viceversa, es decir, que los datos pueden ser cambiados sin afectar a la interfaz de usuario.

En OpenERP, el MVC se implementa de la siguiente forma:

- El **modelo** son las tablas de la base de datos.
- La **vista** son los archivos XML que definen la interfaz de usuario del módulo.
- El **controlador** son los objetos creados en Python.



1

EJEMPLO

En el diagrama anterior, las flechas continuas significan que el controlador tiene un acceso completo a la vista y al modelo, y las flechas discontinuas significan un acceso limitado. Las

razones de este diseño son las siguientes:

- Acceso de **modelo** a **vista**: el modelo envía una notificación a la vista cuando sus datos han sido modificados, con el fin de que la vista pueda actualizar su contenido. El modelo no necesita conocer el funcionamiento interno de la vista para realizar esta operación. Sin embargo, la vista necesita acceder a las partes internas del controlador.
- Acceso de **vista** a **controlador**: la razón de que la vista tenga limitado el acceso al controlador es porque las dependencias que la vista tiene sobre el controlador deben ser mínimas ya que el controlador puede ser sustituido en cualquier momento.

1.2. ESPECIFICACIONES TÉCNICAS PARA EL DESARROLLO DE COMPONENTES

¿Recuerdas en la primera unidad cuando hablábamos de la arquitectura de los sistemas de planificación empresarial? Decíamos que la mayoría están basados en una arquitectura cliente-servidor, este es el caso de OpenERP.

OpenERP utiliza los protocolos XML-RPC o Net-RPC para comunicación entre cliente y servidor. Básicamente lo que hacen es permitir al cliente hacer llamadas a procedimientos remotos, o sea, ejecutar código de otra máquina. En el caso de XML-RPC, la función llamada, sus argumentos y el resultado se envían por HTTP y son codificadas usando XML. Net-RPC está disponible más recientemente, está basado en funciones en Python y es más rápido.

OpenERP funciona sobre un marco de trabajo o framework llamado OpenObject. Este framework permite el desarrollo rápido de aplicaciones (RAD), siendo sus principales elementos los siguientes:

- ORM: mapeo de bases de datos relacionales a objetos Python.
- Arquitectura MVC (Modelo Vista Controlador).
- Diseñador de informes.
- Herramientas de Business Intelligence y cubos multidimensionales.
- Cliente de escritorio y cliente web.

1.3. ESPECIFICACIONES FUNCIONALES PARA EL DESARROLLO DE COMPONENTES

Como sabemos, OpenERP se compone de un núcleo y varios módulos dependiendo de las necesidades:

- **base:** es el módulo básico compuesto de objetos como empresas (res.partner), direcciones de empresas (res.partner.address), usuarios (res.user), monedas (res.currency), etc.
- **account:** gestión contable y financiera.
- **product:** productos y tarifas.
- **purchase:** gestión de compras.
- **sale:** gestión de ventas.
- **mrp:** fabricación y planificación de recursos.
- **crm:** gestión de las relaciones con clientes y proveedores.

1.3.1. CREAR UN MÓDULO

Por cada módulo existe una carpeta con el nombre del módulo en el directorio C:\Archivos de programa\OpenERP AllInOne\Server\addons (del servidor). Para crear un módulo tendremos que crear una carpeta dentro de la carpeta (addons), y seguir los siguientes pasos:

- Crear el archivo de **inicio de módulo**: `__init__.py`
- Crear el archivo con la **descripción del módulo**: `__terp__.py` (hasta la versión 5.0) o `__openerp__.py` (en adelante).

El archivo `__terp__.py` debe contener los siguientes valores dentro de un diccionario Python (estructura de datos cerrada con llaves { }):

- *name*: nombre del módulo.
- *version*: versión del módulo.
- *description*: una descripción del módulo.
- *author*: persona o entidad que ha desarrollado el módulo.
- *website*: sitio web del módulo.
- *license*: tipo de licencia del módulo (por defecto GNU/GPL).


La Licencia Pública General de GNU o más conocida por su nombre en inglés GNU General Public License (o simplemente sus siglas del inglés GNU/GPL) es la licencia más ampliamente usada en el mundo del software y garantiza a los usuarios finales (personas, organizaciones, compañías) la libertad de usar, estudiar, compartir (copiar) y modificar el software. Su propósito es declarar que el software cubierto por esta licencia es software libre y protegerlo de intentos de apropiación que restrinjan esas libertades a los usuarios.

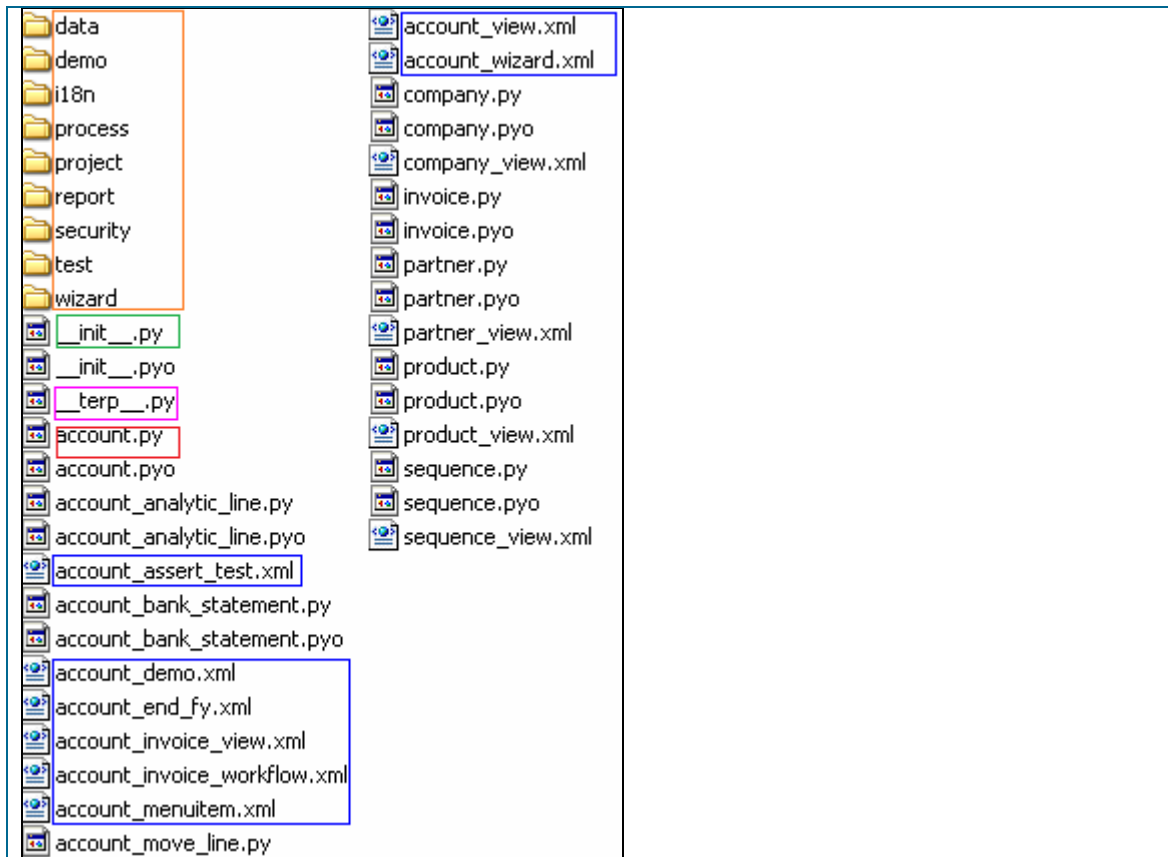
- 2

Abrir con el block de notas el archivo : `__terp__.py` y comprobar su contenido:

- Crear los archivos **Python** con la definición de objetos: **nombre_modulo.py**, por medio de clases del lenguaje Python definimos el modelo y el controlador.
- **Vista o vistas del objeto:** **nombre_modulo_nombre_objeto.xml**.
- **Subcarpetas** **report**, **wizard**, **test**,...: Contienen archivos del tipo de objeto utilizado, en este caso, informes, asistentes y accesos directos, datos de prueba, etc.

Buscar el módulo **account** y comprobar su contenido:

Dirección  C:\Archivos de programa\OpenERP AllInOne\Server\addons\account



Para ver los archivos ocultos y las extensiones de todos los archivos: (Menú) Herramientas → Opciones de carpeta → (Solapa) Ver → Configuración avanzada:

- ☐ Ocultar archivos protegidos del sistema operativo
- ☐ Ocultar las extensiones de archivos para tipo de archivos conocidos

1

EJERCICIO

Buscar un módulo en la carpeta **addons** y anota los siguientes contenidos de dicho módulo:

- Archivo de **inicio de módulo**
- Archivo con la **descripción del módulo**
 - *name:*
 - *version:*
 - *description:*
 - *author:*
 - *website*
 - *license:*
 - *depends:*

- *init_xml:*
- *installable:*
- Archivos **Phyton**
- **Vista o vistas del objeto**
- **Subcarpetas**

2. TÉCNICAS DE OPTIMIZACIÓN DE CONSULTAS Y ACCESO A GRANDES VOLÚMENES DE INFORMACIÓN

Un aspecto a tener en cuenta a la hora de hacer modificaciones en la aplicación y crear módulos que utilicen nuevos objetos, es que estos objetos se basan en consultas a la información existente en la base de datos. Si deseamos mejorar los tiempos de respuesta del sistema, deberemos crear objetos que utilicen consultas lo más optimizadas posibles. Podemos decir que la **optimización** es el proceso de modificar un sistema para mejorar su eficiencia o el uso de los recursos disponibles.

Cuando manejamos grandes cantidades de datos, el resultado de una consulta puede tomar un tiempo considerable, obteniendo no siempre una respuesta óptima. Dentro de las técnicas de optimización de consultas podemos encontrar las siguientes:

- **Diseño de tablas:** A la hora de crear nuevas tablas, asegurarnos de que no hay duplicidad de datos y que se aprovecha al máximo el almacenamiento en las tablas.
- **Campos:** Es recomendable ajustar al máximo el espacio en los campos para no desperdiciar espacio.
- **Índices:** Permiten búsquedas a una velocidad notablemente superior, pero no debemos estar incitados a indexar todos los campos de una tabla, ya que los índices ocupan más espacio y se tarda más al actualizar los datos. Dos de las razones principales para utilizar un índice son:
 - Es un campo utilizado como criterio de búsquedas.
 - Es una clave ajena en otra tabla.
- **Optimizar sentencias SQL:** Existen una serie de reglas para utilizar el lenguaje de consulta y modificación de datos que hay que contemplar. Estas reglas se refieren tanto a la utilización de las sentencias de selección, como a las que realizan alguna inserción o modificación en la base de datos.

- **Optimizar la base de datos:** También podemos conectarnos en modo comando a la base de datos y utilizar sentencias para optimizar los datos contenidos en la base de datos.

2.1. OPERACIONES DE CONSULTA: HERRAMIENTAS

Para optimizar la base de datos necesitamos conectarnos con PostgreSQL en modo comando, los pasos serían los siguientes:

- **Cambiarnos al usuario postgres.** Esto debemos hacerlo porque tenemos que entrar en el monitor interactivo con un usuario que exista en la base de datos PostgreSQL:

```
$ sudo su postgres
```

- **Entramos en el monitor interactivo de PostgreSQL llamado psql con la orden psql:**

```
$ psql
postgres@ubuntu:/home/profesor$ psql

psql (8.4.8)

Digite «help» para obtener ayuda.

postgres=#
```

Una vez dentro del monitor interactivo, el Prompt (postgres=#) significa que el monitor está listo y que podemos escribir comandos.

- **Salir** del monitor de PostgreSQL, con el comando \q.
- Entre los **comandos** que podemos utilizar tenemos:
 - \h: Ayuda.
 - \> Muestra las bases de datos existentes.
 - \c [nombre_bd]: Nos conectamos con la base de datos que queramos.
 - \d: Muestra las tablas existentes en la base de datos.
 - \d [nombre_tabla]: Muestra la descripción de una tabla, o sea, los campos que contiene, de qué tipo son, etc.
 - VACUUM VERBOSE ANALYZE [tabla]; : Limpia y analiza bases de datos.
 - \q: Salimos del editor de consultas.

Crear la base de datos **viviendas**, con las siguientes tablas:

vivienda(cod_vivienda, cod_tipo_vivienda, nombre_vivienda)

PK FK
tipo-vivienda(cod_tipo_vivienda, tipo_vivienda, precio)
 PK

Con los siguientes valores:

vivienda	cod_vivienda	cod_tipo_vivienda	nombre_vivienda	precio
	001	001	La jirafa	85.000
	002	001	Los álamos	95.000
	003	003	La Fresneda	120.000
	004	004	Los balagares	250.000

tipo-vivienda	cod_tipo_vivienda	tipo_vivienda	metros
	001	Piso	100
	002	Atico	150
	003	Chalet adosado	250
	004	Chalet individual	350

- Cambiar al usuario postgres y entrar en el monitos psq:

\$ sudo su postgres

Password: alumno

postgres@alumno-pc: /home/alumno\$ psql

psql (8.4.8)

Digite «help» para obtener ayuda

postgres=#

- Crear la base de datos borrar

postgres=# CREATE DATABASE borrar;

CREATE DATABASE

- Comprobar que se creó esa base de datos

postgres=# \l

- Borrar la base de datos borrar

postgres=# DROP DATABASE borrar;

DROP DATABASE

- Comprobar que se borro esa base de datos

postgres=# \l

- Crear la base de datos **viviendas**

postgres=# CREATE DATABASE viviendas;

CREATE DATABASE

- Conectarse a la base de datos **viviendas**

```
postgres=# \c viviendas;
```

You are now connected to database "viviendas" as user "postgres"

```
viviendas=#
```

- Crear la tabla **tipo_vivienda**

```
viviendas=# CREATE TABLE tipo_vivienda(cod_tipo_vivienda VARCHAR(3), tipo_vivienda VARCHAR(20), metros VARCHAR(4), CONSTRAINT PK_tipo_vivienda PRIMARY KEY(cod_tipo_vivienda));
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "pk_tipo_vivienda" for table "tipo_vivienda"
CREATE TABLE
```

- Crear la tabla **vivienda**

```
viviendas=# CREATE TABLE vivienda(cod_vivienda VARCHAR(3), cod_tipo_vivienda VARCHAR(3), nombre_vivienda VARCHAR(10), precio VARCHAR(9), CONSTRAINT PK_vivienda PRIMARY KEY(cod_vivienda), CONSTRAINT FK_vivienda FOREIGN KEY(cod_tipo_vivienda) REFERENCES tipo_vivienda(cod_tipo_vivienda));
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "pk_vivienda" for table "vivienda"
CREATE TABLE
```

- Mostrar las tablas creadas

```
viviendas=# \d
               List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | tipo_vivienda  | table | postgres
public | vivienda       | table | postgres
(2 rows)
```

- Mostrar la descripción de esas tablas

```
viviendas=# \d tipo_vivienda
Table "public.tipo_vivienda"
  Column          |      Type      | Modifiers
-----+-----+-----
cod_tipo_vivienda | character varying(3) | not null
tipo_vivienda     | character varying(20) |
metros            | character varying(4) |
Indexes:
    "pk_tipo_vivienda" PRIMARY KEY, btree (cod_tipo_vivienda)
Referenced by:
    TABLE "vivienda" CONSTRAINT "fk_vivienda" FOREIGN KEY (cod_tipo_vivienda) REFERENCES tipo_vivienda(cod_tipo_vivienda)

viviendas=# \d vivienda
Table "public.vivienda"
  Column          |      Type      | Modifiers
-----+-----+-----
cod_vivienda      | character varying(3) | not null
cod_tipo_vivienda | character varying(3) |
nombre_vivienda   | character varying(20) |
precio            | character varying(9) |
Indexes:
    "pk_vivienda" PRIMARY KEY, btree (cod_vivienda)
Foreign-key constraints:
    "fk_vivienda" FOREIGN KEY (cod_tipo_vivienda) REFERENCES tipo_vivienda(cod_tipo_vivienda)
```

- Introducir datos en la tabla **tipo_vivienda**

```
viviendas=# INSERT INTO tipo_vivienda (cod_tipo_vivienda, tipo_vivienda, metros)
VALUES ('001', 'Piso', '100');
INSERT 0 1
viviendas=# INSERT INTO tipo_vivienda (cod_tipo_vivienda, tipo_vivienda, metros)
VALUES ('002', 'Atico', '150');
INSERT 0 1
viviendas=# INSERT INTO tipo_vivienda (cod_tipo_vivienda, tipo_vivienda, metros)
VALUES ('003', 'Chalet adosado', '250');
INSERT 0 1
viviendas=# INSERT INTO tipo_vivienda (cod_tipo_vivienda, tipo_vivienda, metros)
VALUES ('004', 'Chalet individual', '350');
INSERT 0 1
```

- Comprobar que se han introducido los datos

```
viviendas=# SELECT * FROM tipo_vivienda;
cod_tipo_vivienda | tipo_vivienda | metros
-----+-----+-----
001                | Piso          | 100
002                | Atico         | 150
003                | Chalet adosado | 250
004                | Chalet individual | 350
(4 rows)
```

- Introducir datos en la tabla **vivienda**

```
viviendas=# INSERT INTO vivienda (cod_vivienda, cod_tipo_vivienda, nombre_vivienda, precio)
VALUES ('001', '001', 'La jirafa', '85000');
INSERT 0 1
viviendas=# INSERT INTO vivienda (cod_vivienda, cod_tipo_vivienda, nombre_vivienda, precio)
VALUES ('002', '001', 'Los álamos', '95000');
INSERT 0 1
viviendas=# INSERT INTO vivienda (cod_vivienda, cod_tipo_vivienda, nombre_vivienda, precio)
VALUES ('003', '003', 'La fresneda', '120000');
INSERT 0 1
viviendas=# INSERT INTO vivienda (cod_vivienda, cod_tipo_vivienda, nombre_vivienda, precio)
VALUES ('004', '004', 'Los balagares', '250000');
INSERT 0 1
```

- Comprobar que se han introducido los datos

```
viviendas=# SELECT * FROM vivienda;
cod_vivienda | cod_tipo_vivienda | nombre_vivienda | precio
-----+-----+-----+-----
001          | 001              | La jirafa       | 85000
002          | 001              | Los álamos      | 95000
003          | 003              | La fresneda     | 120000
004          | 004              | Los balagares   | 250000
(4 rows)
```

- Realizar consultas sobre los datos de ambas tablas
 - Mostrar los nombres de la viviendas

```
viviendas=# SELECT nombre_vivienda FROM vivienda;
nombre_vivienda
-----
La jirafa
Los álamos
La fresneda
Los balagares
(4 rows)
```

- Mostrar los nombres de las viviendas cuyo precio sea 95000

```
viviendas=# SELECT nombre_vivienda FROM vivienda WHERE precio='95000';
nombre_vivienda
-----
Los álamos
(1 row)
```

- Mostrar los metros de los Áticos

```
viviendas=# SELECT metros FROM tipo_vivienda WHERE tipo_vivienda='Atico';
metros
-----
150
(1 row)
```

- Indica para la vivienda 'Los álamos' que tipo de vivienda es:

```
viviendas=# SELECT tipo_vivienda.tipo_vivienda FROM tipo_vivienda, vivienda WHERE vivienda.nombre_vivienda='Los álamos' AND vivienda.cod_tipo_vivienda=tipo_vivienda.cod_tipo_vivienda;
tipo_vivienda
-----
Piso
(1 row)
```

2

EJERCICIO

Crea la siguiente base de datos con las consultas que se indican a continuación:

BASE DE DATOS VIDEOCLUB

PELÍCULAS(codpel, título, duración, tema, precio, clasificación, año, fcompra, país)

SOCIOS(codsocio, nombre, DNI, dirección, tfno, fingresso)

PRESTAMOS(codpel, codsocio, fretirada, fentrega)

1. Consultar el código, nombre y fecha de ingreso en el videoclub de socios con apellido López.
2. Calcular cuántas personas se hicieron socias en el último mes del año 2003.
3. Datos de los socios cuyo teléfono empieza por 22 o por 23 y viven en la avenida de Madrid.
4. Datos de los socios ordenados por la fecha de ingreso.
5. Código de socio, nombre y fecha de ingreso ordenados por la fecha de ingreso en orden ascendente y por el nombre en orden descendente.
6. Calcular cuál es la duración mayor de las películas del videoclub.
7. Consultar los datos de la película más larga.
8. Calcular el precio medio de una película en nuestro videoclub.
9. Calcular cuántas películas están por encima del precio medio.
10. Precio, título y dirección de las películas que están por debajo del precio medio.
11. Préstamos que han tenido una duración superior a dos días.
12. Datos de las películas que han sido prestadas con una duración superior a dos días.
13. Nombres de los socios que han alquilado la película de código 16.
14. Seleccionar las películas ordenadas por país en orden ascendente, tema en orden ascendente y precio descendente.
15. Seleccionar las películas españolas en orden descendente por tema y en orden descendente por clasificación.
16. Socios que han alquilado alguna de las películas prestadas al socio número 8.
17. Consultar cuántas películas ha retirado el socio número 4.
18. ¿Cuántas películas distintas ha retirado Carlos López Cano?
19. Préstamos que aún no han sido devueltos.

PARA SABER MÁS: [Tutorial de PostgreSQ](#)

2.2. SISTEMAS BATCH INPUTS. GENERACIÓN DE PROGRAMAS DE EXTACCIÓN PROCESAMIENTO DE DATOS

El término batch-input procede de la utilización en sistemas SAP (empresa que comercializa un conjunto de aplicaciones de software empresarial) de un método utilizado para transferir grandes cantidades de datos a un sistema ERP. Existen dos formas de hacer un batch-input:

- **Método clásico:** Método asíncrono, es decir, que se procesan los datos pero se actualizan más tarde. Tiene la característica de que genera un archivo de mensajes o log para tratarse errores a posteriori.
- **Método "call transaction":** Método on-line usado para dar de alta rápidamente pocos registros. Es un método síncrono, no genera log es mucho más rápido pero poco útil para gran cantidad de datos.

Un proceso batch-input se compone de dos fases:

- **Fase de generación:** Es la fase en que se genera el archivo batch-input con los datos a introducir o modificar.
- **Fase de procesamiento** El archivo batch-input se ejecuta, haciéndose efectivas las modificaciones en la base de datos.

3. LENGUAJE PROPORCIONADO POR LOS SISTEMAS ERP-CRM

La mayoría de aplicaciones de planificación empresarial están construidas con lenguajes de programación modernos y, en muchos casos, orientados a objetos. Existen aplicaciones ERP que proporcionan lenguajes propietarios, que no son lenguajes estándar y requieren que el programador se forme en ese lenguaje sólo y exclusivamente para manejar el ERP.

El lenguaje de programación utilizado en OpenERP es un lenguaje relativamente reciente, el lenguaje Python.

Este lenguaje fue creado por Guido van Rossum a principios de los años 90 en el Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica), en los Países Bajos. Guido sigue siendo el autor principal de Python.

En 2001, se creó la Python Software Foundation (PSF), una organización sin fines de lucro creada específicamente para poseer la propiedad intelectual sobre la licencia y promover el uso del lenguaje. Como miembros patrocinadores de esta organización están Google, Canonical, Sun Microsystems y Microsoft.

La licencia de Python es de código abierto, compatible con GPL. Es un lenguaje muy sencillo de utilizar con una sintaxis muy cercana al lenguaje natural, por lo que se considera uno de los mejores lenguajes para empezar a programar.

Python es utilizado por Google, Yahoo, la NASA o por ejemplo en todas las distribuciones Linux, donde los programas escritos en este lenguaje representan un porcentaje cada vez mayor. Aunque ver en profundidad este lenguaje nos llevaría más tiempo del esperado, vamos a ver algunas de sus características más importantes para poder crear nuestros propios módulos en OpenERP.

PARA SABER MÁS: [Lenguaje Python](#)

3.1. CARACTERÍSTICAS E INTRODUCCIÓN AL LENGUAJE

Las principales características de Python son:

- **Sintaxis muy sencilla:** lo cual facilita un rápido aprendizaje del lenguaje.
- **Lenguaje interpretado.** Los programas en Python se ejecutan utilizando un programa intermedio llamado intérprete. El código fuente se traduce a un archivo llamado bytecode con extensión .pyc o .pyo, que es el que se ejecutará en sucesivas ocasiones.
- **Tipado dinámico:** Esta característica quiere decir que los datos no se declaran antes de utilizarlos, sino que el tipo de una variable se determina en tiempo de ejecución, según el dato que se le asigne.

- **Fuertemente tipado:** Cuando una variable es de un tipo concreto, no se puede usar como si fuera de otro tipo, a no ser que se haga una conversión de tipo. Si la variable es numérica entera, por ejemplo, no podré asignarle un valor real. En otros lenguajes, el tipo de la variable cambiaría para adaptarse al comportamiento esperado, aunque ello puede dar más lugar a errores.
- **Multiplataforma:** El intérprete de Python está disponible para una gran variedad de plataformas: Linux, Windows, Macintosh, etc.
- **Orientado a objetos:** Los programas están formados por clases y objetos, que son representaciones del problema en el mundo real. Los datos, junto con las funciones que los manipulan, son parte interna de los objetos y no están accesibles al resto de los objetos. Por tanto, los cambios en los datos de un objeto sólo afectan a las funciones definidas para ese objeto, pero no al resto de la aplicación.

3.1.1. **INSTALAR PYTHON**

3.1.1.1. **En Linux**

```
$ sudo apt-get install python
```

```
$ python (Para ejecutar Python)
```

Muestra un mensaje con la versión que tenemos instalada y el prompt `>>>`, que indica que está esperando código del usuario.

3.1.1.2. **En Windows XP**

Instalar python-3.3.0

A partir de este momento las pruebas se harán en Windows XP: en modo comando o en el intérprete de Python

3.1.2. **HACER UNA PRUEBA en modo comando**

5	EJEMPLO
Inicio → Programas → Python 3.3 → IDLE(Python GUI)	
Para hacer el primer programa en Python no necesitamos conocer mucho sobre el lenguaje, por ejemplo, podemos hacer el famoso "Hola mundo" con el que se suele comenzar a estudiar casi todos los lenguajes, simplemente con la siguiente línea:	
<pre>> print('Hola mundo')</pre>	
Hola mundo	

3.1.3. HACER UNA PRUEBA en el intérprete de Python

6	EJEMPLO
<p>Inicio → Programas → Python 3.3 → IDLE(Python GUI) → (Menú) File → New Window</p> <pre>print('Hola mundo')</pre> <p>(Menú) File → Save (Lo guarda con extensión .py o .pyw)</p> <p>(Menú) Run → Run Module (Para ejecutarlo)</p> <p>Hola mundo</p>	

3.1.4. SALIR

Podemos salir escribiendo `exit()`, o con la combinación de teclas Control + D.

3.1.5. MAS COSAS

Los comentarios en el lenguaje se escriben con el carácter `#`, las asignaciones de valores a las variables con el carácter igual (`=`). Para crear una función utilizamos el siguiente código, hay que tener en cuenta que la sangría del texto sirve para delimitar qué instrucciones van dentro de la función:

7	EJEMPLO
<pre>def nombrefunción(arg1,arg2..): instrucción1 instrucción2 ... instrucciónN</pre>	

Es decir, en este lenguaje no hay delimitador de sentencia como el punto y coma de Java o C, ni delimitador de bloques, como las llaves de estos mismos lenguajes.

PARA SABER MÁS: [Descargar intérprete de Python](#)

3.2. DECLARACIÓN DE DATOS**3.2.1. TIPOS DE DATOS BÁSICOS**

Como ya hemos comentado, en Python no se declaran las variables como tal, sino que se utilizan directamente. Podemos declarar variables de los siguientes tipos de datos básicos:

- **Números enteros:** Los números enteros son el cero y aquellos números positivos o negativos que no tienen decimales. En Python se pueden representar mediante el tipo int que utiliza 32 o 64 bits, dependiendo del tipo de procesador, o el tipo long, que permite almacenar números de cualquier precisión, estando limitados solo por la memoria disponible en la máquina.

8

EJEMPLO

```
# declaración de un número entero

a = 2

print(a)

2

# declaración de un número entero largo

a = 2L
```

- **Números reales:** Son los que tienen la coma decimal. Se expresan mediante el tipo float.

9

EJEMPLO

```
b = 2.0

print(a, b)

2 2.0
```

- **Números complejos:** Son un tipo de dato especial compuesto por una parte real y una parte imaginaria.

10

EJEMPLO

```
complejo = 2.0 + 7.5 j
```

- **Cadenas:** Es texto encerrado entre comillas simples o dobles.

11

EJEMPLO

```
mensaje = "Bienvenido a Python"

print(mensaje)

Bienvenido a Python
```

- **Booleanos:** Sólo pueden tener dos valores True o False.

•

12

EJEMPLO

```
c=True
```

```
print(c)
```

```
True
```

3

EJERCICIO

Crea los siguientes tipos de datos: Número entero, numero real, cadena y booleano. Y mostrar en pantalla el contenido de cada uno de ellos.

3.2.2. OPERADORES

De entre los operadores más importantes para trabajar con estos datos podemos destacar los siguientes:

- **Operadores aritméticos:** suma (+), resta (-), multiplicación (*), división(/), división entera (//), exponente (**), módulo (%).

13

EJEMPLO

```
a=10
```

```
b=2
```

```
c=3
```

```
suma=a+b
```

```
print("La suma es:", suma)
```

```
La suma es: 12
```

```
resta=a-b
```

```
print("La resta es:", resta)
```

```
La resta es: 8
```

```
multiplicacion=a*b
```

```
print("La multiplicación es:", multiplicacion)
```

```
La multiplicación es: 20
```

```
division=a/b
```

```
print("La division es:", division)
```

```
La division es: 5.0
```

```
division=a/c
```

```

print("La division es:", division)

La division es: 3.3333333333333335

divisionentera=a//c

print("La divisionentera es:", divisionentera)

La divisionentera es: 3

division=b/c

print("La division es:", division)

La division es: 0.6666666666666666

divisionentera=b//c

print("La divisionentera es:", divisionentera)

La divisionentera es: 0 (NO redondea, simplemente elimina los decimales)

exponente=b**c

print("El exponente es:", exponente)

El exponente es: 8

resto=a%c

print("El resto es:", resto)

El resto es: 1

```

3

EJERCICIO

Mostrar en pantalla la tabla de multiplicar del número 5.

SOLUCION

```

numero=5

resultado1=numero*1

resultado2=numero*2

.....

Resultado10=numero*10

print(numero, '* 1 =' ,resultado1, '\n',numero, '* 2 =' ,resultado2, ..... , '\n',numero, '*
10 =' ,resultado10)

5 * 1 = 5

5 * 2 = 10

```

$$5 * 3 = 15$$

.....

$$5 * 10 = 50$$

- **Operadores booleanos:** and, or, not.

14

EJEMPLO

Mostrar en pantalla los valores de la siguiente tabla:

a	b	and
0	0	0
0	1	0
1	0	0
1	1	1

SOLUCION

resultado1=0 and 0

resultado2=0 and 1

resultado3=1 and 0

resultado4=1 and 1

```
print('0 and 0 =',resultado1,'\n','0 and 1 =',resultado2,'\n','1 and 0 =',resultado3,'\n','1
and 1 =',resultado4,'\n')
```

0 and 0 = 0

0 and 1 = 0

1 and 0 = 0

1 and 1 = 1

4

EJERCICIO

Mostrar en pantalla los valores de la siguiente tabla:

a	b	or
0	0	0
0	1	1
1	0	1
1	1	1

SOLUCION

resultado1=0 or 0

resultado2=0 or 1

```
resultado3=1 or 0
```

```
resultado4=1 or 1
```

```
print('0 or 0 =',resultado1,'\n','0 or 1 =',resultado2,'\n','1 or 0 =',resultado3,'\n','1 or 1 =',resultado4,'\n')
```

```
0 or 0 = 0
```

```
0 or 1 = 1
```

```
1 or 0 = 1
```

```
1 or 1 = 1
```

5

EJERCICIO

Mostrar en pantalla los valores de la siguiente tabla:

a	not
0	1
1	0

SOLUCION

```
resultado1=not 0
```

```
resultado2=not 1=10
```

```
print('not 0 =',resultado1,'\n','not 1 =',resultado2)
```

```
not 0 = 1
```

```
not 1 = 0
```

- **Operadores relacionales:** igual (==), distinto (!=), menor (<), mayor (>), menor igual(<=), mayor o igual (>=).

15

EJEMPLO

Dadas las siguientes variables:

```
a=10
```

```
b=2
```

```
c=2
```

Realizar las siguientes comparaciones:

```
a==b
```

```
False
```

```
a!=b
```

`True``a<b``False``a>b``True``a<=b``False``b<=c``True``a>=b``True``b>=c``True`

6

EJERCICIO

Crea diferentes variables, realiza **operaciones aritméticas** con ellas y comprueba los resultados utilizando los **operadores relacionales**.

3.3. ESTRUCTURAS DE PROGRAMACIÓN: COLECCIONES

Python tiene una serie de estructuras de datos que se utilizan ampliamente. Son tipos avanzados de datos, que reciben el nombre de Colecciones y son:

- **Listas:** Son colecciones ordenadas de datos, en otros lenguajes se conocen como arrays o vectores. Pueden tener cualquier tipo de dato.

16

EJEMPLO

```
l = [3, True, "mi lista", [ 1 , 2 ]]
```

Se accede a un elemento concreto de la lista usando el índice de su posición (empezando por cero) entre corchetes.

```
a = l[0]    # a vale 3
```

```
print(a)
```

```
3
```

```
b = l[3][1]    # b vale 2
```

```
print(a)
```

```
2
```

7

EJERCICIO

Crea la siguiente lista y muestra por pantalla sus elementos ordenados de la siguiente forma:

a[i][j]	00	01	02	03
	10	11	12	13
	20	21	22	23
	30	31	32	33

SOLUCION

```
a=[['00','01','02','03'],['10','11','12','13'],['20','21','22','23'],['30','31','32','33']]
```

```
print(a[0][0],a[0][1],a[0][2],a[0][3],'\n',a[1][0],a[1][1],a[1][2],a[1][3],'\n',
```

```
a[2][0],a[2][1],a[2][2],a[2][3],'\n',a[3][0],a[3][1],a[3][2],a[3][3])
```

```
00 01 02 03
```

```
10 11 12 13
```

```
20 21 22 23
```

```
30 31 32 33
```

- **Tuplas:** Son parecidas a las listas, con la diferencia que son más ligeras por lo que si el uso es básico se utilizan mejor tuplas. Los elementos van entre paréntesis en lugar de entre corchetes.

17

EJEMPLO

```
t = ( "hola", 2, False)
```

```
b = t[1] # b vale 2
```

```
print(b)
```

```
2
```

Las listas son objetos mutables, es decir, podemos modificar cada uno de sus componentes. Las tuplas sin embargo son no mutables, es decir, que no podemos modificarlas una vez que se han creado, por ejemplo, si intentamos hacer esto:

18

EJEMPLO

```
lista_semanal = ["Lunes", "Martes", "miércoles", "Jueves", "Viernes", "Sabado",  
"Domingo"]
```

```
lista_semanal[2]= "Miércoles"
```

```
tupla_semanal= ("Lunes", "Martes", "miércoles", "Jueves", "Viernes", "Sabado",  
"Domingo")
```

```
tupla_semanal[2]= "Miércoles"
```

Traceback (most recent call last):

File "<pyshell#12>", line 1, in <module>

```
tupla_semanal[2]= "Miércoles"
```

TypeError: 'tuple' object does not support item assignment

El intérprete de Python nos da un error porque las tuplas no se pueden modificar

- **Diccionarios:** Son colecciones que relacionan una clave y un valor. Para acceder a un valor se utiliza el operador [], igual que para las filas y tuplas, y dentro de él la clave a la que queremos acceder.

19

EJEMPLO

```
d = { "Nombre": "Alberto",
```

```
"Apellido" : "Rodriguez",
```

```
"Victorias" : [ 2007 , 2008 , 2009 ] }
```

```
x = d ["Nombre"]    # x vale "Alberto"
```

```
print(x)
```

```
Alberto
```

```
x = d["Apellido"]   # x vale "Rodriguez"
```

```
print(x)
```

```
Rodriguez
```

```
x = d["Victorias"][1]  # x vale 2008
```

```
print(x)
```

```
2008
```

3.4. SENTENCIAS DEL LENGUAJE

La sintaxis de las estructuras de programación en Python es la siguiente:

- **Estructura condicional:** La sintaxis más común de la estructura condicional es un `if` seguido de la condición a evaluar, seguida de dos puntos, y la siguiente línea sangrada, para indicar el código que queremos que se ejecute si se cumple la condición. En caso de que no se cumpla la condición, ejecutamos la sentencia detrás del `else`:

20

EJEMPLO

Realizar un programa **par.py** para calcular si un número es par o impar.

Inicio → Programas → Python 3.3 → IDLE(Python GUI) → (Menú) File → New Window

```
numero=input("Introduce un número")
```

```
if (numero % 2 == 0):
```

```
    print ("El numero es par")
```

```
else:
```

```
    print ("El numero es impar")
```

(Menú) File → Save: par.py

(Menú) Run → Run Module (Para ejecutarlo)

Impar

8

EJERCICIO

Realizar un programa llamado **menor** que compare 3 variables para averigüe cuál de ellas es la menor. Dichas variables deberán ser introducidas por teclado con un menú similar al siguiente:

Introduce número1 en v1:

Introduce número2 en v2:

Introduce número3 en v3:

A continuación evaluar las variables.

Ejemplo:

Valores a introducir		
v1	v2	v3
1	2	3
2	1	3
3	2	1
1	3	2
3	1	2
2	3	1

→
→
→
→
→
→
→

Visualizará
La menor es v1=1
La menor es v2=1
La menor es v3=1
La menor es v1=1
La menor es v2=1
La menor es v3=1

```
v1=input("Introduce número1 en v1:")
```

```
v2=input("Introduce número1 en v2:")
```

```
v3=input("Introduce número1 en v3:")
```

```
if((v1<v2)and(v1<v3)):
```

```
    print("La menor es v1=",v1)
```

```
else:
```

```
    if((v2<v1)and(v2<v3)):
```

```
        print("La menor es v2=",v2)
```

```
    else:
```

```
        if((v3<v1)and(v3<v2)):
```

```
            print("La menor es v3=",v3)
```

```
    else:
```

```
        print("Las tres son iguales")
```

- **Bucles:** Nos permiten ejecutar un fragmento de código un número de veces, o mientras se cumpla una determinada condición.
 - El bucle *while* (mientras) repite todo su bloque de código mientras la expresión evaluada sea cierta. Detrás de la condición hay que poner dos puntos, y debajo las sentencias que van dentro del while.

21

EJEMPLO

Realizar un programa llamado **numerosWhile** que permita mostrar los 10 primeros números naturales a través del bucle while.

```
numero=1
```

```
while (numero <= 10):
```

```

print(numero)

numero +=1

1
.....
10

```

El funcionamiento es sencillo, el programa evalúa la condición, y si es cierta, ejecuta el código que hay dentro del while, cuando acaba vuelve a evaluar la condición y si es cierta ejecuta nuevamente el código, y así hasta que la condición devuelva false, en cuyo caso, el programa continuaría ejecutando las instrucciones siguientes

9

EJERCICIO

Realizar un programa llamado **sumaNumerosWhile** que permita mostrar la suma de los 10 primeros números naturales a través del bucle while.

```

numero=1

suma=0

while (numero <= 3):

    suma=suma+numero

    numero +=1

print("La suma es:", suma)

La suma es: 55

```

- El bucle *for* se utiliza para recorrer secuencias de elementos.

23

EJEMPLO

Realizar un programa llamado **numerosFor** que permita mostrar los 10 primeros números naturales a través del bucle for.

<pre> lista_numeros=[1,2,3,4,5,6,7,8,9,10] for numero in lista_numeros: print(numero) 1 10 </pre>	<pre> for i in range(10): print(i+1) 1 10 i varia de 0 a 9 </pre>
--	--

Lo que hace el bucle es que para cada elemento que haya en la secuencia, ejecuta las líneas de código que tenga dentro, en este caso la sentencia print. Lo que hace la cabecera del bucle es obtener el siguiente elemento de la secuencia lista_numeros y almacenarlo en una variable de nombre numero. Por esta razón en la primera iteración del bucle numero valdrá 1, en la segunda 2, en la tercera 3....

24

EJEMPLO

Realizar un programa llamado **letrasFor** que permita mostrar las 10 primeras letras del alfabeto a través del bucle for.

```
lista_letras=["a","b","c","d","e","f","g","h","i","j"]
```

```
for letra in lista_letras:
```

```
    print(letra)
```

```
a
```

```
.....
```

```
j
```

10

EJERCICIO

Realizar un programa llamado **sumaNumerosFor** que permita mostrar la suma de los 10 primeros números naturales a través del bucle for.

```
lista_numeros=[1,2,3,4,5,6,7,8,9,10]
```

```
suma=0
```

```
for numero in lista_numeros:
```

```
    suma=suma+numero
```

```
print("La suma es:",suma)
```

```
La suma es: 55
```

```
suma=0
```

```
for numero in range(10):
```

```
    suma=suma+(numero+1)
```

```
print("La suma es:",suma)
```

```
La suma es: 55
```

3.5. LLAMADAS A FUNCIONES

Una función es un conjunto de instrucciones o programa que realiza una tarea específica y que devuelve un valor. Las funciones nos ayudan a reutilizar código, y a dividir el programa en partes más pequeñas con objeto de hacerlo más organizado, legible y más fácil de depurar.

25

EJEMPLO

Realizar un programa llamado **cuadrado** en el que a través de una función realice el cuadrado

del número que se le pasa a la función como parámetro.

Declarar una función `cuadrado` que, calcula el cuadrado de un numero.

```
def cuadrado ( numero ):
```

```
    print ("El cuadrado de 4 es:", numero ** 2)
```

El número que aparece entre paréntesis es el valor que le pasamos a la función para que pueda calcular su cuadrado, y recibe el nombre de parámetro. Las funciones pueden tener uno, varios o ningún parámetro.

Cuando ejecutamos la función se dice que estamos haciendo una llamada a la función, y lo hacemos utilizando su nombre y los parámetros entre paréntesis.

Llamada a la función:

```
cuadrado (4)
```

```
16
```

NOTA: Escribir todo este código en el archivo **cuadrado.py** y en ese orden

Las funciones pueden o no devolver un valor como resultado de su ejecución. Podemos transformar el ejemplo anterior para que devuelva el resultado en lugar de mostrarlo por pantalla. El código sería el siguiente:

26

EJEMPLO

Declarar una función

```
def cuadrado ( numero ):
```

```
    return numero ** 2
```

Llamada a la función:

```
print ("El cuadrado de 4 es:", cuadrado(4))
```

```
16
```

El resultado sería el mismo, sólo que ahora la función devuelve el valor, y ese valor es el que muestra por pantalla la instrucción `print`

11

EJERCICIO

Realizar un programa llamado **suma** que cree una función para sumar 2 números introducidos por teclado. Utilizar una función `suma` para realizar la suma de dichos números.

Hacer el ejercicios de las dos formas vistas anteriormente.

1ª forma:

Declarar una función

```
def suma(numero1, numero2):  
    print ("La suma es:", numero1 + numero2)
```

Llamada a la función:

```
numero1=3  
numero2=5  
suma (numero1, numero2)  
  
8
```

2ª forma:

Declarar una función

```
def suma(numero1, numero2):  
    return numero1 + numero2
```

Llamada a la función:

```
numero1=3  
numero2=5  
def suma(numero1, numero2):  
    return numero1 + numero2  
  
Llamada a la función:  
numero1=3  
numero2=5  
print("La suma es:", suma (numero1, numero2)  
  
8
```

3.6. CLASES Y OBJETOS

Hasta ahora hemos visto la sintaxis básica de Python. Pero para entender bien cómo está formado un módulo en OpenERP, necesitamos saber también qué son las clases y los objetos. Si has programado antes con un lenguaje orientado a objetos, no tendrás ningún problema en entender la definición de ambos conceptos.

Un programa orientado a objetos está formado por clases y objetos. Cuando tenemos un problema que resolver, lo dividimos en una serie de objetos y son esos objetos los que

describimos en nuestro programa. El programa consiste en una serie de interacciones entre esos objetos. Las clases son plantillas a partir de las cuales se crean los objetos, cuando creamos un objeto a partir de una clase se dice que ese objeto es una instancia de la clase.

Las clases tienen una serie de atributos o campos que las definen, así como una serie de métodos para hacer operaciones sobre esos campos. Cuando creamos un objeto le damos valor a esos atributos.

3.6.1. DEFINIR LA CLASE

En Python las clases se definen mediante la palabra clave `class` seguida del nombre de la clase, dos puntos (`:`) y a continuación, sangrado, el cuerpo de la clase.

27	EJEMPLO
<p>Crear un programa llamado personas en el que se va a crear una clase persona que tiene como atributos el nombre y la edad:</p> <pre>class persona: def __init__(self, nombre, edad): self.nombre = nombre self.edad = edad def mayordeedad(self): if self.edad > 18: print ("Es mayor de edad") else: print ("No es mayor de edad")</pre>	

En la definición de clase anterior vemos que estamos definiendo un método `__init__`. El nombre de este método se utiliza para inicializar el objeto.

Cuando instanciamos un objeto a partir de la clase, será el primer método que se ejecutará. También vemos que los dos métodos de la clase tienen como primer argumento `self`. Este argumento sirve para referirnos al objeto actual, y se utiliza para poder acceder a los atributos y métodos del objeto.

3.6.2. CREAR UN OBJETO de la clase

28	EJEMPLO
<p>Para crear un objeto de la clase utilizamos la siguiente instrucción:</p> <pre>empleado = persona("Javier",32)</pre>	

3.6.3. ACCEDER LOS ATRIBUTOS Y MÉTODOS del objeto

Ahora que ya hemos creado nuestro objeto, podemos acceder a sus atributos y métodos mediante la sintaxis objeto.atributo y objeto.método:

29

EJEMPLO

```
print (empleado.nombre, "tiene", empleado.edad)
```

12

EJERCICIO

Crear un programa llamado **figuras** en el que se creará una clase llamada figura y darle como atributos Ancho y Alto

```
class figura:
```

```
    ancho = 20
```

```
    alto = 40
```

Este código de por si no hace absolutamente nada. Solo crea un tipo de objeto llamado figura, pero hasta que no se cree una instancia de este objeto no pasa nada.

Para crearla una instancia de figura, lo tenemos que hacer por medio de una variable:

```
caja = figura ()
```

Ya tenemos nuestra primera instancia. Ahora vamos a comprobar si funciona bien.

```
print (caja.alto)
```

```
40
```

Todo esto se ha hecho en el mismo archivo. Pero que pasa si queremos separar las clases. Creamos un nuevo archivo, que este en la misma carpeta que en el que estábamos trabajando, que lo vamos a llamar figuras.

```
from figuras import figura
```

```
caja = figura ()
```

```
print caja.Alto
```

3.7. MÓDULOS Y PAQUETES

Cuando salimos del intérprete y volvemos a entrar empezamos desde el principio, si habíamos creado alguna variable o programa éste desaparece y tenemos que volver a escribir el código. Para conservar los programas que hacemos, podemos crear archivos con extensión .py, de manera que siempre tengamos el código disponible para ejecutarlo con el intérprete y que no sea necesario volver a introducirlo.

Por otra parte, los programas a veces son muy grandes y al dividirlos en partes se hacen más fáciles de mantener. Se agrupa el código relacionado y se guarda en archivos diferentes. Cada archivo es un módulo en Python. En otras palabras, un módulo es un archivo que contiene definiciones y declaraciones de Python.

Las razones para utilizar módulos son principalmente las siguientes:

- Hacer el código más fácil de mantener.
- Reutilización de código, por ejemplo, una función que queramos utilizar en varios programas.

3.7.1. SENTENCIA IMPORT

Se utiliza para utilizar un módulo en otro archivo. Importamos el módulo con la siguiente instrucción:

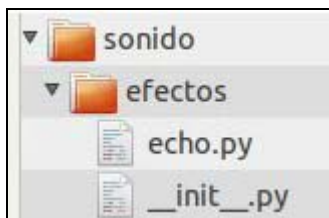
```
import nombre_modulo
```

Podemos importar:

- Módulos individuales.
- Subcarpetas.
- O paquetes completos.

30

EJEMPLO



Importar:

- Módulos individuales: `import sonido.efectos.echo`
- Subcarpetas: `import sonido.efectos`
- O paquetes completos: `import sonido`

Para hacer que Python trate a un directorio como un paquete es necesario crear un archivo `__init__.py` en dicha carpeta. Para hacer que un módulo pertenezca a un paquete hay que copiar el archivo `nombre_modulo.py` que define el módulo en el directorio del paquete.

13

EJERCICIO

Crear un módulo **funciones.py** que defina una función que escriba el texto "Bienvenido a

Python". Utilizar el módulo en otro que se llame **mi_modulo.py**.

SOLUCIÓN

Para crear el módulo lo único que tenemos que hacer es crear un archivo y meter dentro el código de la función, que tal y como hemos visto se define de la siguiente forma:

funciones.py

```
def bienvenido():  
    print("Bienvenido a Python")  
  
bienvenido()
```

Hay que tener cuidado con las sangrías, porque en Python es la manera de indicar que una instrucción está dentro de un bloque de código, y puede dar error de ejecución si no están bien definidas.

Utilizar este módulo en el archivo **mi_modulo.py**:

mi_modulo.py

```
import funciones
```

Se ejecuta:

Bienvenido a Python

3.7.2. DIFERENCIA ENTRE MÓDULOS Y PAQUETES

Módulo: Archivo que contiene definiciones y declaraciones en Python. Ese archivo es del tipo `nombre_modulo.py`.

Paquete: Es una colección de módulos relacionados. Es decir, son las carpetas que incluyen un archivo, y son del tipo `__init__.py`

Mientras los módulos ayudan a organizar el código, los paquetes ayudan a organizar los módulos.

3.8. LIBRERIAS DE FUNCIONES (APIs)

Una API, o Biblioteca de Clases, es un conjunto de clases útiles que tienen a disposición los programadores para utilizar en sus programas. Python proporciona una API estándar, cuyo código fuente está a libre disposición para las principales plataformas desde el sitio web de

Python. También es posible acceder a muchos módulos libres de terceros en la misma página web, así como a programas, herramientas y documentación adicional.

A continuación se indican algunas de las librerías de Python:

- **os**: Provee de diferentes funciones del sistema operativo, como creación de archivos y directorios, leer o escribir de un archivo, manipular rutas, etc.
- **sys**: Permite acceder a información sobre el intérprete de Python, como por ejemplo el prompt del sistema, datos de licencia y en general cualquier parámetro o variable que tenga que ver con el intérprete.
- **date, time**: Provee módulos para la manipulación de fechas y horas.
- **math**: Provee funciones matemáticas.

3.8.1. SENTENCIA IMPORT

Se utiliza para importar una librería: `import nombre_libreria`

31

EJEMPLO

Crear un programa **hora.py** que muestre la hora en los siguientes formatos:

- En segundos
- Como tupla
- En representación textual

```
import time #Importa librería time
```

```
print("La hora en segundos es:",time.time())
```

```
print("La hora como tupla es:", time.localtime())
```

```
print("La hora en representación textual es:", time.asctime())
```

```
La hora en segundos es: 1359049991.902197
```

```
La hora como tupla es: time.struct_time(tm_year=2013, tm_mon=1, tm_mday=24, tm_hour=18,
tm_min=53, tm_sec=11, tm_wday=3, tm_yday=24, tm_isdst=0)
```

```
La hora en representación textual es: Thu Jan 24 18:53:11 2013
```

Si comentamos `import time` con `#` → El programa daría un error porque no encuentra la librería `time` que contiene esas funciones

PARA SABER MÁS: *Ejemplo de programa que utiliza los paquetes `os` y `time` para obtener información del sistema:*

32

EJEMPLO

Crear un programa para obtener la información del sistema que ejecuta el script; cosas como el PID del script, el directorio donde se está ejecutando, el nombre y versión del sistema operativo, etc.

SOLUCION

```
#!/usr/bin/<SPAN><SPAN>p</SPAN>ython</SPAN>
```

```
import os
```

```
import time
```

```
#numUsuario = os.getuid()
```

```
pidProceso = os.getpid()
```

```
donde = os.getcwd()
```

```
#sistemaOperativo = os.uname()
```

```
tiempos = os.times()
```

```
horaRaw = time.time()
```

```
horaFormato = time.ctime(horaRaw)
```

```
#print ("Numero de usuario",numUsuario)
```

```
print ("PID",pidProceso)
```

```
print ("Directorio actual",donde)
```

```
#print ("Informacion del sistema",sistemaOperativo)
```

```
print ("Informacion de tiempos del sistema",tiempos)
```

```
print ("\nLa hora/fecha actual es",horaRaw)
```

```
print ("Lo cual significa",horaFormato)
```

```
PID 3028
```

```
Directorio actual C:\Python33
```

```
Informacion de tiempos del sistema nt.times_result(user=0.03125, system=0.21875,  
children_user=0.0, children_system=0.0, elapsed=0.0)
```

```
La hora/fecha actual es 1360151015.515625
```

```
Lo cual significa Wed Feb 6 12:43:35 2013
```

PARA SABER MÁS: [Biblioteca de clases estándar de Python](#)

Elimina los comentarios del EJEMPLO 32 y con la ayuda del enlace: [PARA SABER MÁS: Biblioteca de clases estándar de Python](#) corrige los errores que muestra el programa al ejecutar dicho ejemplo

3.9. INSERCIÓN, MODIFICACIÓN Y ELIMINACIÓN DE DATOS EN OBJETOS (APIs)

En este apartado vamos a ver cómo están estructurados los módulos y cómo crear un primer módulo sencillo que cree un objeto en la aplicación para insertar, modificar o eliminar datos en la base de datos.

Existen 2 formas de crear un módulo:

3.9.1. 1ª FORMA DE CREAR UN MÓDULO EN OPENERP

Los módulos de OpenERP se guardan dentro de la carpeta addons.

Lo que tenemos que hacer es crear una carpeta para nuestro módulo dentro del directorio addons. Los archivos que tiene que contener la carpeta son:

- **__init__.py**: Necesario para que la carpeta se trate como un paquete en Python, contiene los import de cada archivo del módulo que contenga código Python, en este caso, el archivo nombre_modulo.py.
- **__terp__.py**: Contiene un diccionario Python con la descripción del módulo, como quién es el autor, la versión del módulo o cuáles son los otros módulos de los que depende.
- **nombre_modulo.py**: En este archivo creamos la clase definiendo los campos que va a tener. Al crear la clase estamos creando el modelo (una tabla en la base de datos) y también estamos creando el controlador, porque cuando creamos una clase estamos definiendo el comportamiento que tiene.
- **nombre_modulo_view.xml**: En este archivo definimos la vista de nuestro módulo, o del objeto que va a crear nuestro módulo. Para crear este archivo necesitamos tener ciertos conocimientos de XML, o bien coger una vista de otro objeto y a partir de ella crear la del nuevo objeto.

La arquitectura MVC (Modelo Vista Controlador) sirve para separar los datos, de la lógica de negocio y de la interfaz de usuario, OpenERP sigue esta arquitectura, el modelo son las tablas de postgres, la vista son los XML que definen la interfaz de usuario del módulo y el Controlador son los Objetos creados en python.

Vamos a crear un módulo **coche** que permita gestionar en OpenERP una lista de matriculas y modelo de coches.

- El módulo que vamos a crear se llamará coche → Crear la siguiente carpeta: C:\Archivos de programa\OpenERP AllInOne\Server\addons\coche, donde se guardarán los archivos mínimos para este módulo y son los siguientes:

- __init__.py**: Define los atributos y métodos del paquete, en este caso solo contendrá los import de los ficheros y directorios que contengan código.

```
import coche
```

- __terp__.py**: Contiene la descripción del modulo:

```
{
    "name" : "coche",
    "author" : "Angel",
    "version" : "0.1",
    "depends" : ["base"],
    #esta es la lista de los módulos de los que depende
    "init_xml" : [],
    "update_xml" : ["coche_view.xml"],    #aquí indicamos la vista
    "category" : "Enterprise Specific Modules/Coches",
    "active": False,
    "installable": True
}
```

- coche.py**: En coche.py tenemos tanto el controlador como el modelo, tenemos el controlador porque se trata de una clase python en la que definimos el comportamiento, y tenemos el modelo porque esta clase hereda de la clase osv (Object Service) que implementa una capa de ORM (Object-Relational mapping), para los que venimos de java lo que sería hibernate. Así las clases que creamos heredando de osv y siguiendo esta estructura crean las tablas en la base de datos.

Como vemos importamos osv para que coche herede de el, e importamos también fields para definir el tipo de las columnas, en este ejemplo solo hemos usado cadenas de caracteres, ya veremos mas tipos de columnas.

```
from osv import osv, fields
class coche(osv.osv):
    _name = 'coche'
    _columns = {
        'modelo': fields.char('Modelo', size=64),
        'matricula': fields.char('Matricula', size=64),
    }
coche()
```

- coche_view.xml**: En el se define la vista, en este ejemplo se ha creado la entrada para el menú y la definición de la ventana listado y formulario.

```
<openerp>
```

```
<data>

<record model="ir.actions.act_window" id="action_coche_form">

    <field name="res_model">coche</field>

    <field name="domain">[]</field>

</record>

<menuitem name="Coches" id="menu_coche_coche_form"/>

<menuitem name="Coches" id="menu_coche_form" parent="coche.menu_coche_coche_form"
action="action_coche_form"/>

<record model="ir.ui.view" id="view_coche_form">

    <field name="name">coche</field>

    <field name="model">coche</field>

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

    <field name="priority" eval="5"/>

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

        <form string="Coche">

            <field name="modelo" select="1"/>

            <field name="matricula" select="1" />

        </form>

    </field>

</record>

<record model="ir.ui.view" id="view_coche_tree">

    <field name="name">coche</field>

    <field name="model">coche</field>

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

    <field name="priority" eval="5"/>

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

        <tree string="Coche">

            <field name="modelo" select="1"/>

            <field name="matricula" select="1" />

        </tree>

    </field>

</record>
```


</field>

</record>

</data>

</openerp>

- En la carpeta C:\Archivos de programa\OpenERP AllInOne\Server\addons\coche tendremos los siguientes ficheros:

__init__.py

__terp__.py

coche.py

coche_view.xml

- Abrir la aplicación OpenERP → Crear una base de datos:
 - Nombre de la base de datos: bd_coche
 - ☐ Cargar datos de demostración
 - Contraseña administrador: admin1
 - Perfil: Minimal profile
 - Modo de vista: Interfaz Extendida
- Cargar el módulo coche:
 - Menú → Administración → Administración de módulos → (2 Clic) Actualizar lista de módulos
 - Menú → Administración → (2 Clic) Módulos → (2 Clic) coche → Programar para instalación → Aplicar actualizaciones programadas
 - Formulario → Refrescar
- Introducir datos en la tabla coche:
 - Menú → Coches → (2 Clic) coches → Nuevo:
 - Modelo: Audi
 - Matricula: 11111111
- Consultar datos en la tabla coche:
 - Menú → Coches → (2 Clic) coches

Modificar el ejercicio anterior creando una nueva base de datos **bd_coche_modificada**, cambiando el Autor por tu nombre y añadiendo el campo color a la tabla coche.

- **__init__.py**

```
import coche
```

- **__terp__.py**

```
{
    "name" : "coche",
    "author" : "Mercedes",
    "version" : "0.1",
    "depends" : ["base"],
    #esta es la lista de los módulos de los que depende
    "init_xml" : [],
    "update_xml" : ["coche_view.xml"],    #aquí indicamos la vista
    "category" : "Enterprise Specific Modules/Coches",
    "active": False,
    "atallable": True
}
```

- **coche.py**

```
from osv import osv, fields
class coche(osv.osv):
    _name = 'coche'
    _columns = {
        'modelo': fields.char('Modelo', size=64),
        'matricula': fields.char('Matricula', size=64),
        'color': fields.char('Color', size=34),
    }
coche()
```

- **coche_view.xml**

```
<openerp>
<data>

    <record model="ir.actions.act_window" id="action_coche_form">
        <field name="res_model">coche</field>
        <field name="domain">[]</field>
    </record>

    <menuitem name="Coches" id="menu_coche_coche_form"/>
    <menuitem name="Coches" id="menu_coche_form" parent="coche.menu_coche_coche_form">
```

```
action="action_coche_form"/>

<record model="ir.ui.view" id="view_coche_form">

    <field name="name">coche</field>

    <field name="model">coche</field>

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

    <field name="priority" eval="5"/>

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

        <form string="Coche">

            <field name="modelo" select="1"/>

            <field name="matricula" select="1" />

            <field name="color" select="1" />

        </form>

    </field>

</record>

<record model="ir.ui.view" id="view_coche_tree">

    <field name="name">coche</field>

    <field name="model">coche</field>

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

    <field name="priority" eval="5"/>

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

        <tree string="Coche">

            <field name="modelo" select="1"/>

            <field name="matricula" select="1" />

            <field name="color" select="1" />

        </tree>

    </field>

</record>

</data>
```

3.9.2. 2ª FORMA DE CREAR UN MÓDULO EN OPENERP (con el módulo *base_module_record*)

También existe otra forma de crear módulos sin necesidad de ningún desarrollo, que es utilizando el módulo *base_module_record*. Si instalamos este módulo, tendremos la opción de grabar todas las acciones que realicemos en la aplicación. Si has utilizado alguna vez las macros en un procesador de textos o en una hoja de cálculo sabrás de lo que estamos hablando. El módulo *base_module_record* funciona de forma parecida. Por ejemplo, podemos crear un nuevo objeto y asignarle un menú, todo ello se grabaría en un archivo comprimido. Realmente ese archivo comprimido lo que va a contener es un módulo, con todos los archivos indicados anteriormente. Si con posterioridad cogemos esos archivos y los copiamos a la carpeta addons, podremos instalarlos en la aplicación como cualquier otro módulo. Al instalarlo, el resultado sería que aparecería ese nuevo objeto y menú en nuestra aplicación.

PARA SABER MÁS: [Crear un módulo en OpenERP con el módulo *base_module_record*](#)

4. ENTORNOS DE DESARROLLO Y HERRAMIENTAS DE DESARROLLO EN SISTEMAS ERP-CRM

La herramienta más importante cuando estamos programando siempre es un buen entorno de desarrollo. Si queremos una respuesta rápida y el programa es pequeño podemos utilizar el intérprete, como hemos venido haciendo hasta ahora. Sin embargo, cuando el programa va siendo más grande, es mucho más cómodo utilizar un entorno de desarrollo desde el cual podemos hacer todas las operaciones de manera gráfica.

Un entorno de desarrollo es un programa que incluye todo lo necesario para programar en uno o varios lenguajes. Además de un editor de textos especializado como herramienta central, incluyen navegadores de archivos, asistentes de compilación, depuradores, etc.

Entre los entornos de desarrollo para trabajar con Python nos encontramos con uno de los más sencillos, IDLE, disponible en la mayoría de las plataformas. En Ubuntu se puede descargar desde Synaptic. Este entorno básicamente lo que permite es revisar la sintaxis de nuestro módulo y ejecutarlo. También incluye opciones de depuración.

Cuando lo ejecutamos nos aparece una ventana con un sencillo menú. Desde él podremos abrir el archivo con código python, y aparecerá una nueva ventana donde podremos revisar la sintaxis desde el menú Run/Check module o ejecutar el programa desde el menú Run/Run module, entre otras opciones.

En otro orden de cosas, tenemos la extensión o plugin de OpenERP para el editor Gedit. Esta extensión permite utilizar el editor de texto como un entorno de desarrollo para OpenERP. La

principal ventaja es que añade fragmentos de código típico, lo cual facilita en gran medida no tener que recordar la sintaxis.

Finalmente, dentro de los entornos de desarrollo más completos para trabajar con Python tenemos Eclipse. Eclipse además incorpora una serie de plantillas con código, lo cual facilita mucho la creación de los módulos en OpenERP.

4.1. DEPURACIÓN DE UN PROGRAMA

Las herramientas de depuración cuando estamos desarrollando un programa ayudan a detectar errores en el código, mostrando información de qué tipo de error es o porqué se produce. Las herramientas de depuración suelen formar parte de los entornos de desarrollo.

El lenguaje Python incorpora un depurador dentro de su Biblioteca de módulos estándar llamado pdb. Soporta puntos de ruptura y ejecución paso a paso del código, inspeccionar valores de variables y otras opciones de depuración. Sin embargo, como en todos los lenguajes, si utilizamos una herramienta gráfica para el proceso de depuración, seguramente ahorraremos mucho tiempo.

Dentro del entorno de desarrollo IDLE podemos depurar nuestros programas. Este entorno está disponible para cualquier distribución y sistema operativo. Para activar el módulo de depuración hacemos clic en el menú Debug/Debugger. Una vez que esté el modo de depuración activado el funcionamiento es como el de cualquier depurador. Con el botón derecho establecemos una parada en la sentencia del código que queremos evaluar, de manera que al ejecutar el programa el depurador se parará en esa instrucción, y podremos analizar el valor de las variables en ese punto.

Las acciones que podemos realizar con el depurador son las siguientes:

- **Go:** hace que la ejecución del programa continúe hasta que encuentre un punto de ruptura.
- **Step:** va ejecutando el programa línea a línea.
- **Over:** hace que la sentencia actual sea completamente ejecutada sin parar en ninguna función anidada.
- **Out:** hace que se compute desde el punto actual hasta el fin de la función actual y que esta se termine.
- **Quit:** finaliza la ejecución del programa.

4.2. MANEJO DE ERRORES

Cuando nuestro programa falla, se genera un error en forma de excepción. Las excepciones son errores detectados por el intérprete durante la ejecución del programa. Por ejemplo,

podemos intentar dividir por cero o acceder a un archivo que no existe, entonces el programa genera una excepción avisando al usuario de que se ha producido un error. Nuestra labor como programadores es escribir el código de manera que se contemplen esas posibles excepciones, y se actúe en consecuencia. Si nuestro programa no contempla el manejo de excepciones, cuando ocurra alguna lo que pasará es que se generará el error y el programa dejará de ejecutarse o no tendrá el funcionamiento esperado.

En Python se utiliza una construcción try-except para capturar y tratar las excepciones. Dentro del bloque try se sitúa el código que creemos que podría producir una excepción, y dentro del bloque except, escribimos el código que se ejecutará si se produce dicha excepción.

Veamos un pequeño programa que lanzaría una excepción al intentar hacer una división entre 0:

```
def dividir(a, b):  
    return a / b  
  
dividir(3,0)
```

Si lo ejecutamos en el intérprete obtenemos la siguiente salida de error:

```
ZeroDivisionError: integer division or modulo by zero
```

Lo que nos dice el mensaje es que es una excepción, ZeroDivisionError, y la descripción del error: "integer division or modulo by zero" (módulo o división entera entre cero).

Si escribimos el mismo programa, pero teniendo en cuenta el manejo de excepciones, el código quedaría como sigue:

```
try:  
    def dividir(a, b):  
        return a / b  
  
    dividir(1,0)  
except:  
    print ("Ha ocurrido un error")
```

5. FORMULARIOS E INFORMES EN SISTEMAS ERP-CRM

Los **formularios** constituyen la interfaz de un módulo. Una vez creado el objeto del módulo, o sea, el archivo nombre_modulo.py, lo que debemos hacer es crear los menús, acciones, vistas (formulario u otro tipo) para poder interactuar con el objeto. Estos elementos se describen en el archivo nombre_modulo_view.xml.

Decir que es importante recordar que el nombre del fichero XML se encuentra listado en el atributo `update_xml` del fichero `__terp__.p`, `__openerp__.py` a partir de la versión 5.0, del módulo.

Los **informes** pueden ser estadísticos, para los que tenemos el módulo `base_report_creator` que permite crear listados, informes y gráficos por pantalla, y pueden ser informes impresos, en cuyo caso existen diversas herramientas para generarlos.

5.1. ARQUITECTURA DE FORMULARIOS E INFORMES: ELEMENTOS

Los formularios y demás vistas de la aplicación son construidos de forma dinámica por la descripción XML de la pantalla del cliente. Podemos decir que el lenguaje XML es un metalenguaje que utiliza etiquetas del tipo `etiqueta_contenido_fin` etiqueta para expresar información estructurada.

Una etiqueta consiste en una marca hecha en el documento, que hace referencia a un elemento o pedazo de información. Las etiquetas tienen la forma `<nombre>`, donde `nombre` es el nombre del elemento que se está señalando.

En apartados anteriores has tenido acceso a un breve manual sobre el metalenguaje XML. Si lo necesitas puedes echarle un vistazo para comprender mejor los conceptos de este apartado.

En el caso de OpenERP, la estructura del archivo `nombre_modulo_view.xml` es la siguiente:

```
<?xml version="1.0" encoding="utf-8"?>

<openerp>

<data>

<record model="object_model_name" id="object_xml_id">

<field name="field1">value1</field>

<field name="field2">value2</field>

</record>

<record model="object_model_name2" id="object_xml_id2">

<field name="field1" ref="module.object_xml_id"/>

<field name="field2" eval="ref('module.object_xml_id')"/>

</record>

</data>

</openerp>
```

Cada tipo de registro, hace referencia a un objeto diferente. Los registros se definen con la etiqueta `<record>` `</record>`, que indican el inicio y fin de la descripción del registro. Dentro van

los campos field que se utilizan para definir las características del registro. Siguiendo la estructura anterior, un ejemplo de creación de vista de tipo formulario sería el siguiente:

```
<record model="ir.ui.view" id="view_agenda_form">
<field name="name">agenda</field>
<field name="model">agenda</field>
<field name="type">form</field>
<field name="priority" eval="5"/>
<field name="arch" type="xml">
<form string="Agenda">
<field name="nombre" select="1"/>
<field name="telefono" select="1" />
</form>
</field>
</record>
```

El campo name hace referencia al nombre de la vista y el campo name="model" hace referencia al objeto del modelo, es decir, la tabla en la base de datos, del cual va a mostrar la información. Vemos que la vista está formada por dos campos: nombre y telefono.

De igual forma podemos definir el elemento de menú para acceder al objeto agenda:

```
<menuitem name="Agenda" id="menu_agenda_agenda_form"/>
<menuitem
name="Agenda"
id="menu_agenda_form"
parent="agenda.menu_agenda_agenda_form" action="action_agenda_form"/>
```

Así como la acción asociada a ese elemento de menú:

```
<record model="ir.actions.act_window" id="action_agenda_form">
<field name="res_model">agenda</field>
<field name="domain">[]</field>
</record>
```

5.2. HERRAMIENTAS PARA LA CREACIÓN DE FORMULARIOS E INFORMES

Existen diferentes herramientas para la creación de formularios e informes en OpenERP. Para empezar hay que distinguir entre la creación de formularios y la creación de informes. La

creación de formularios como ya hemos visto se realiza en archivos XML con el nombre `nombre_modulo_view.xml`. La creación de informes se realiza con otras herramientas. Ya hemos visto cómo crear informes con el módulo `base_report_creator` y con la extensión de OpenOffice.org para OpenERP. Existe otra forma de crear informes, y es mediante la utilización de la librería Jasper Reports.

Jasper Reports es una librería para la generación de informes. Su funcionamiento consiste en generar un archivo XML con la descripción del informe a crear, que es tratado para generar un documento en un formato de salida, como por ejemplo PDF o HTML.

TAREAS

VER: [tarea1](#)

VER: [tarea2](#)

VER: [tarea3](#)

VER: [tarea4](#)

VER: [tarea5](#)

VER: [tarea6](#)

VER: [tarea7](#)

VER: [tarea8](#)

VER: [tarea9](#)

VER: [tarea2Solución](#)

VER: [tarea3Solución](#)

VER: [tarea4Solución](#)

VER: [tarea5Solución](#)

VER: [tarea6Solución](#)

VER: [tarea7Solución](#)

VER: [tarea8Solución](#)

VER: [tarea9Solución](#)