



Biblioteca Virtual FP

Plan FP 2015

# ERP-CRM IFC04CM15

Juan Félix Mateos

[jfmateos@educa.madrid.org](mailto:jfmateos@educa.madrid.org)

[juanfelixmateos@gmail.com](mailto:juanfelixmateos@gmail.com)

# TEMA 11

## Personalización y desarrollo de módulos a medida para OpenERP

# Introducción a Python

- Python es un lenguaje interpretado (no compilado) de muy alto nivel
- Actualmente existen 2 versiones estables del lenguaje de programación Python:
  - Python 3.4
  - Python 2.7 (es la que se utiliza para OpenERP)
- Python puede descargarse de <http://www.python.org/downloads>
- Para escribir programas en Python simplemente necesitamos un editor de textos.
  - El propio Python incluye un entorno de desarrollo shell muy útil llamado IDLE
  - No obstante, existen entornos más avanzados como:
    - PythonWin
    - EasyEclipse for Python

# Introducción a Python

## Ejecución de programas

- Para ejecutar un programa python utilizaremos la sentencia:
  - `python miprograma.py` o bien directamente `miprograma.py` si tenemos python añadido a la ruta del sistema
  - En Ubuntu, para ejecutar un programa directamente escribiendo su nombre:
    - Debemos configurarlo como ejecutable (Propiedades>Privilegios)
    - Debemos incluir una línea especial con la ruta del intérprete llamada shebang
      - `#!/usr/bin/env python`
    - No podemos escribir directamente el nombre del programa, sino `./miprograma.py`

# Introducción a Python

## Cuestiones léxicas específicas

- No existe un caracter concreto (como el ; en JavaScript) que marque el final de una sentencia. El final de línea se señala con un **salto de línea**
- Los espacios (sangrados) en Python son esenciales
- Los comentarios pueden ser:
  - Una sola línea iniciados por #
  - Varias líneas iniciados y terminados por 3 comillas simples o dobles

# Introducción a Python

## Un programa sencillo

```
#!/usr/bin/env python  
nombre="juanfe"  
if nombre=='juanfe':  
    print 'Hola Juanfe'  
else:  
    print "No te conozco"
```

# Introducción a Python

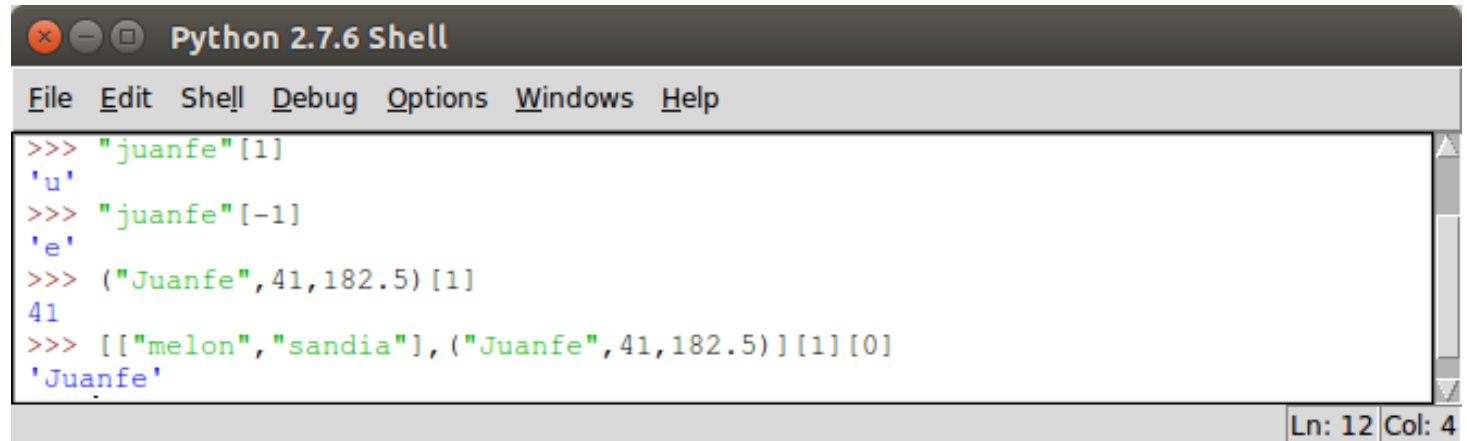
## Tipos de datos

- En Python no es necesario declarar el tipo de las variables (adquieren tipo al asignarles un valor)
- Booleanos: True y False
- Enteros: 41
- Coma flotante: 182.5
- Secuencias (se pueden recorrer y trocear)
  - Cadenas de caracteres (inmutables): "Juanfe"
  - Listas (mutables): ["Juanfe",41,182.5]
  - Tupla (inmutables): ("Juanfe",41,182.5)
- Diccionarios (mutables): {"nombre":"Juanfe","edad":41}

# Introducción a Python

## Recorrer y trocear secuencias

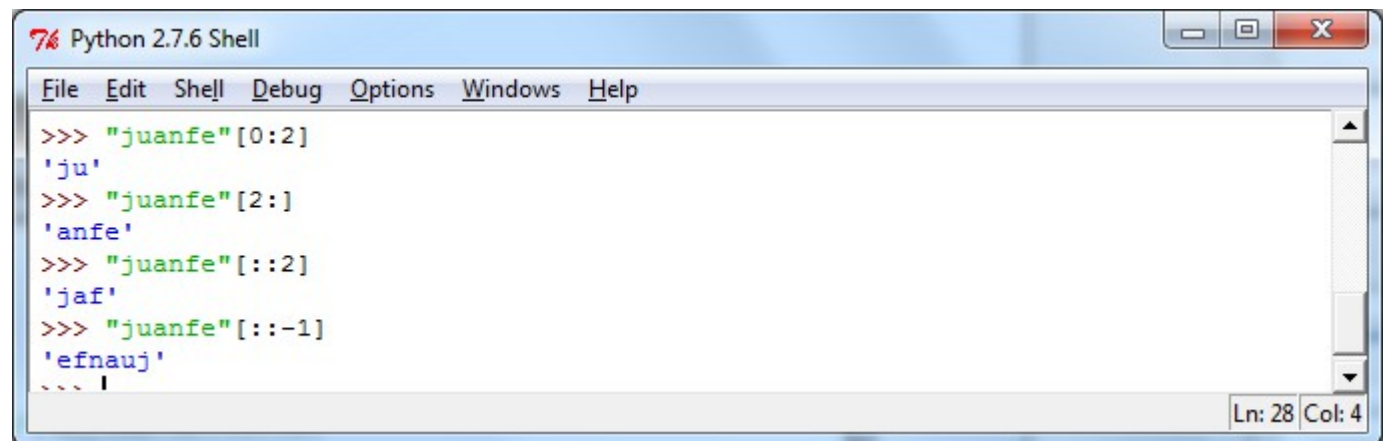
- Recorrer []



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
>>> "juanfe"[1]
'u'
>>> "juanfe"[-1]
'e'
>>> ("Juanfe", 41, 182.5)[1]
41
>>> [ ["melon", "sandia"], ("Juanfe", 41, 182.5) ][1][0]
'Juanfe'
```

Ln: 12 Col: 4

- Trocear [inicio:fin:incremento]



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
>>> "juanfe"[0:2]
'ju'
>>> "juanfe"[2:]
'anfe'
>>> "juanfe"[:2]
'ja'
>>> "juanfe"[:-1]
'efnauj'
```

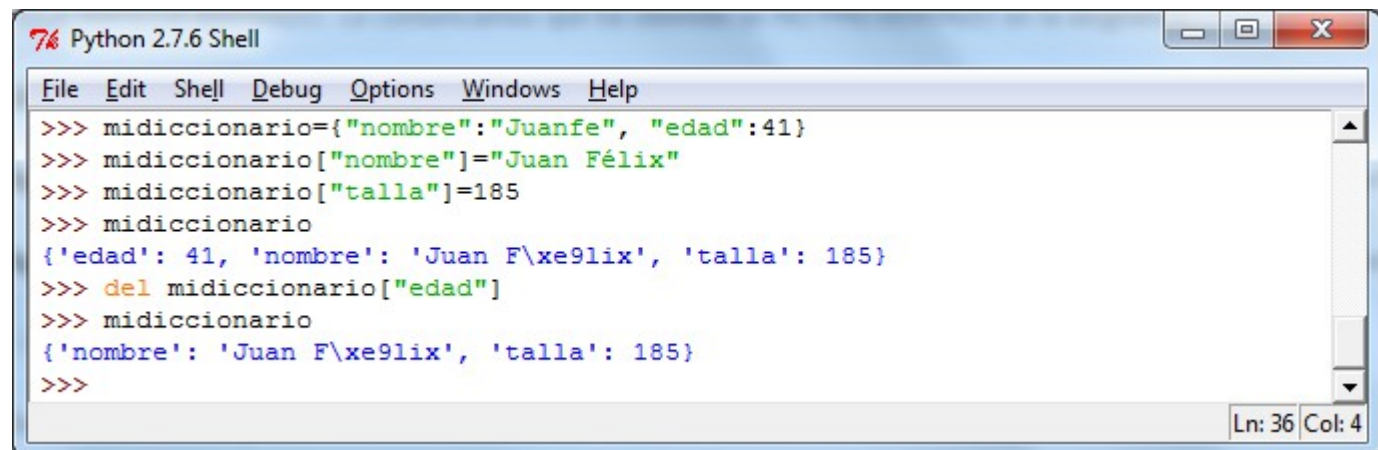
Ln: 28 Col: 4



# Introducción a Python

## Diccionarios

- Los diccionarios son colecciones de pares nombre:valor.
- Los elementos están identificados por sus nombre (no existen índices que impliquen un orden)
- Añadir un elemento: `diccionario["nombre"]=valor`
- Eliminar un elemento: `del diccionario["nombre"]`



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
>>> midiccionario={"nombre":"Juanfe", "edad":41}
>>> midiccionario["nombre"]="Juan Félix"
>>> midiccionario["talla"]=185
>>> midiccionario
{'edad': 41, 'nombre': 'Juan F\x9elix', 'talla': 185}
>>> del midiccionario["edad"]
>>> midiccionario
{'nombre': 'Juan F\x9elix', 'talla': 185}
>>>
```

Ln: 36 Col: 4

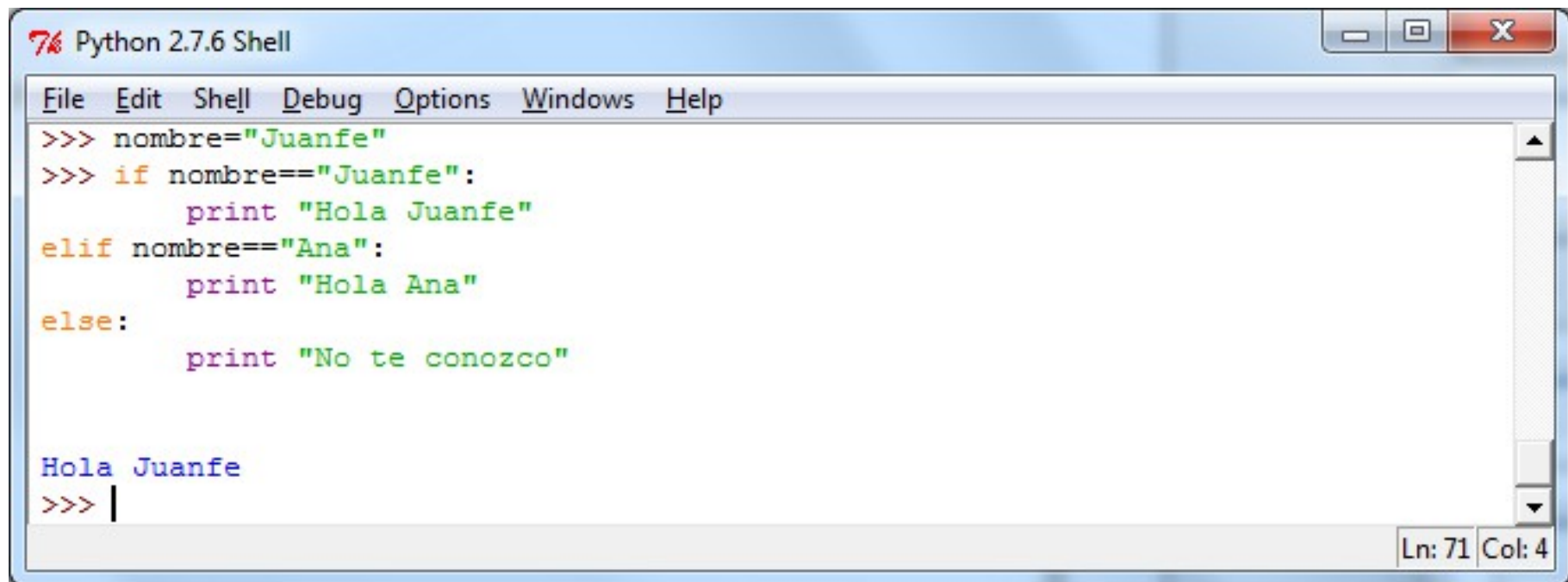
# Introducción a Python

## Operadores

Operador	Descripción	Operador	Descripción
+	Suma y concatenación	in	Busca un valor en una secuencia
-	Resta	<<, >>, &,  , ^, ~	Operadores binarios
*	Producto	=, +=, -=, *=, /=, //=, **=, %=	Operadores de asignación
**	Potencia	<	¿Menor que?
/	División	<=	¿Menor o igual que?
//	Suelo del cociente	>	¿Mayor que?
%	Módulo (resto)	>=	¿Mayor o igual que?
and	Y lógico	==	¿Igual?
or	O lógico	!=	¿Distinto?
not	No lógico	<>	¿Distinto?

# Introducción a Python

## Bifurcación if...elif...else



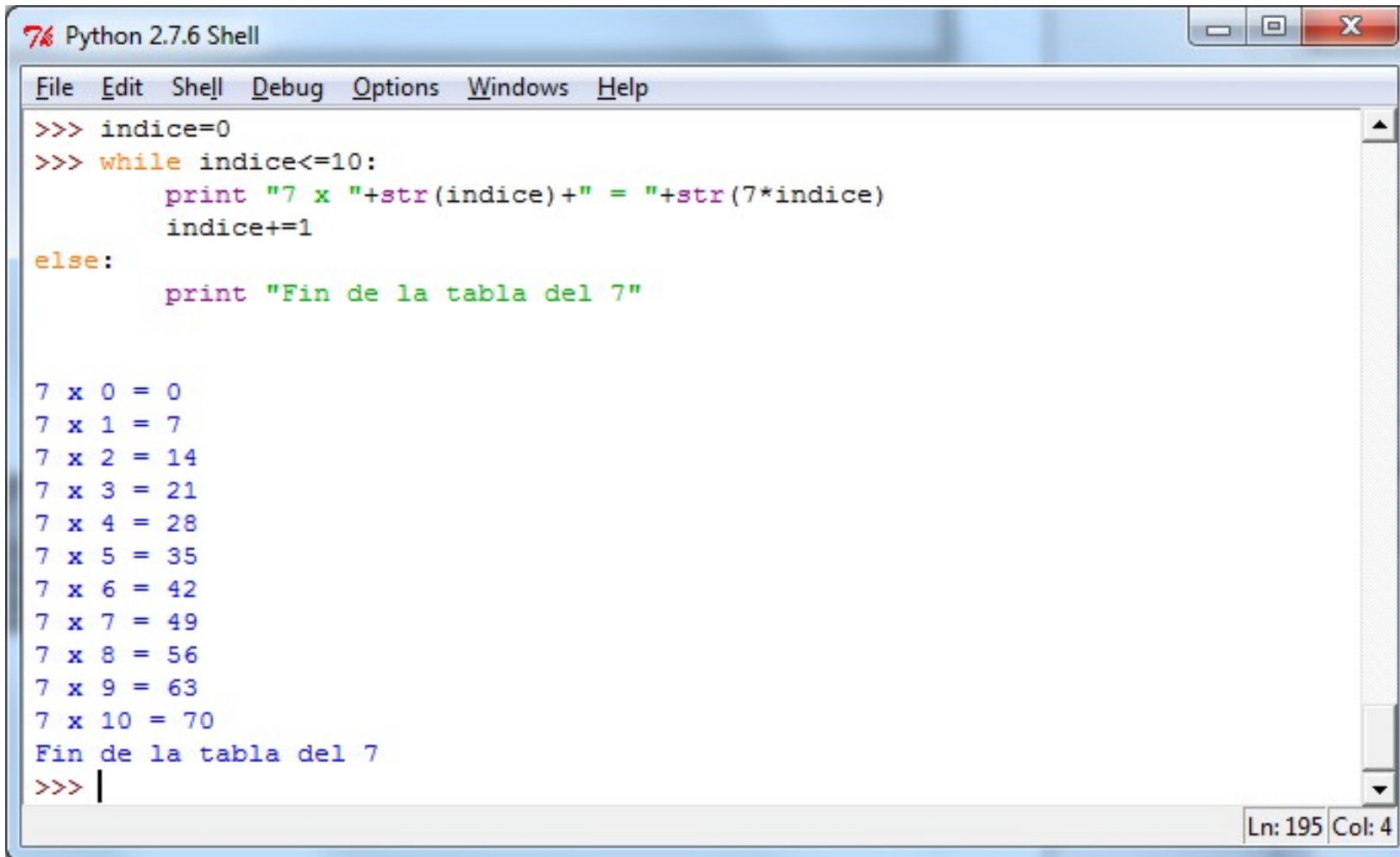
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
>>> nombre="Juanfe"
>>> if nombre=="Juanfe":
        print "Hola Juanfe"
elif nombre=="Ana":
        print "Hola Ana"
else:
        print "No te conozco"

Hola Juanfe
>>> |
```

Ln: 71 Col: 4

# Introducción a Python

## Bucles while...else



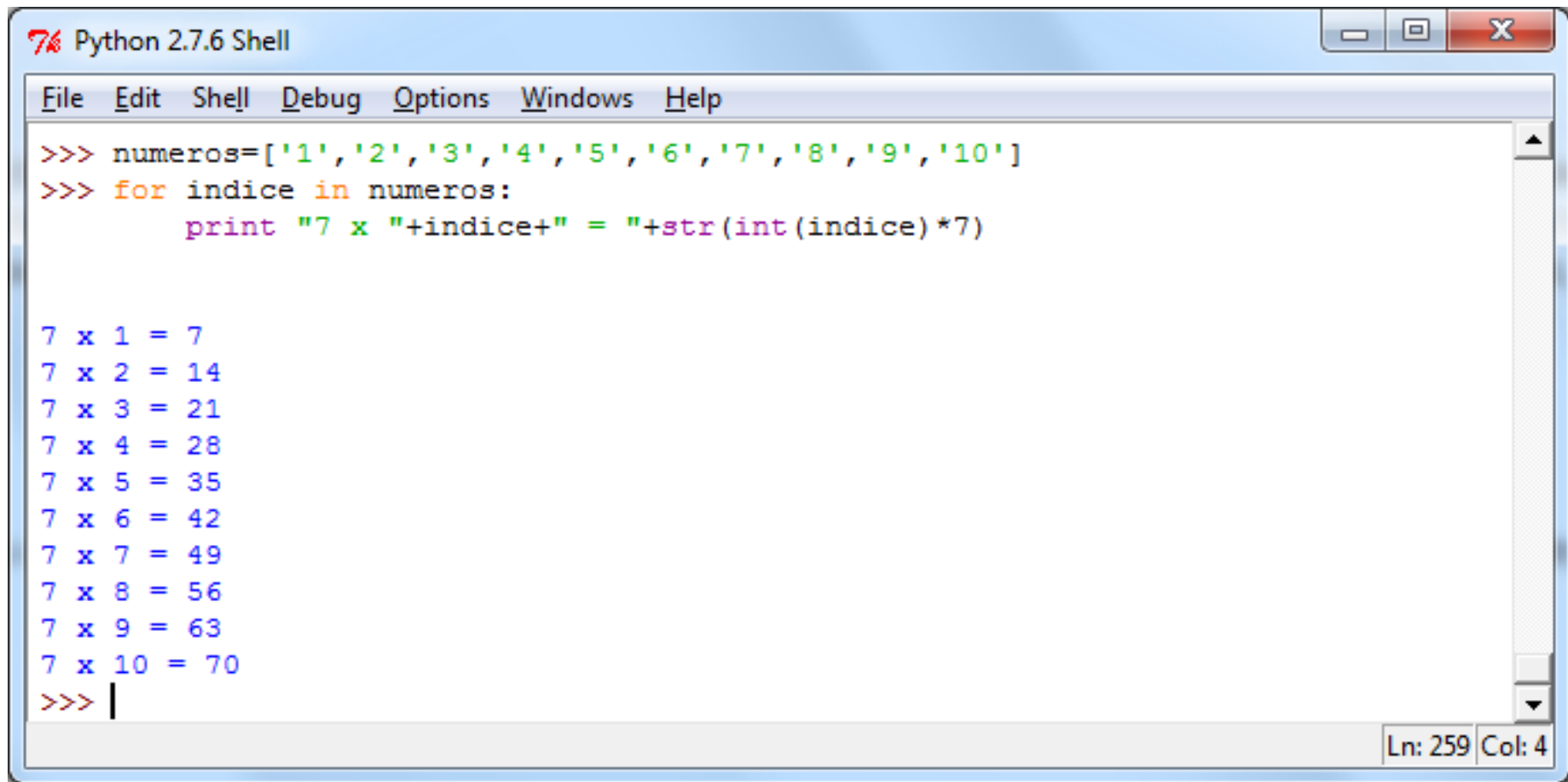
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
>>> indice=0
>>> while indice<=10:
    print "7 x "+str(indice)+" = "+str(7*indice)
    indice+=1
else:
    print "Fin de la tabla del 7"

7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
Fin de la tabla del 7
>>> |
```

Ln: 195 Col: 4

# Introducción a Python

## Bucles for...in



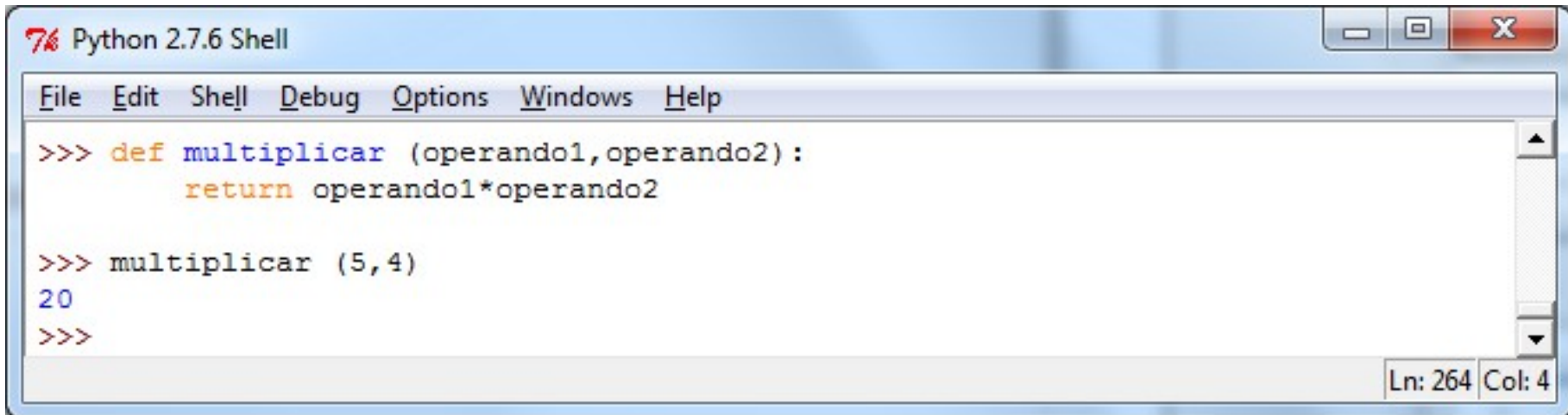
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
>>> numeros=['1','2','3','4','5','6','7','8','9','10']
>>> for indice in numeros:
    print "7 x "+indice+" = "+str(int(indice)*7)

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
>>> |
```

Ln: 259 Col: 4

# Introducción a Python

## Funciones



A screenshot of a Python 2.7.6 Shell window. The window has a title bar with the text "Python 2.7.6 Shell" and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with the following items: File, Edit, Shell, Debug, Options, Windows, and Help. The main area of the window contains a Python code snippet. The code defines a function named "multiplicar" that takes two arguments, "operando1" and "operando2", and returns their product. The function is then called with the arguments 5 and 4, resulting in the output 20. The status bar at the bottom right of the window shows "Ln: 264 Col: 4".

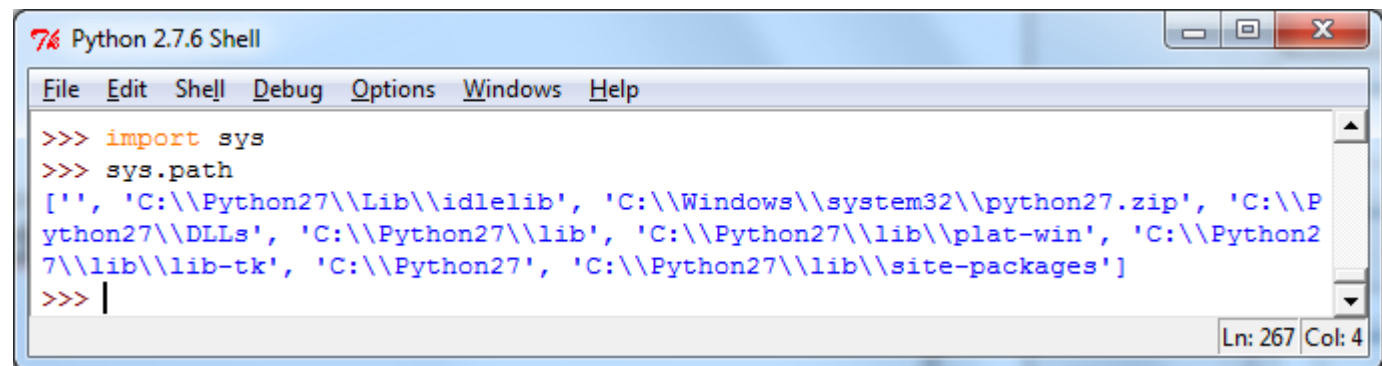
```
>>> def multiplicar (operando1,operando2):  
    return operando1*operando2  
  
>>> multiplicar (5,4)  
20  
>>>
```

Ln: 264 Col: 4

# Introducción a Python

## Módulos

- En Python cada archivo es un módulo, cuyo nombre es el del propio archivo.
- Para importar un módulo dentro de otro podemos hacerlo:
  - Totalmente con: **import** modulo\_a\_importar
  - Parcialmente con: **from** modulo\_a\_importar **import** funcion\_a\_importar
- No es necesario indicar la ruta de acceso al módulo porque Python dispone de una cadena de búsqueda de módulos (ni tampoco la extensión .py):
  - Primero busca en la propia carpeta
  - Luego en las ubicaciones indicadas en la variable de entorno PYTHONPATH
  - Por último en el directorio donde se encuentran las bibliotecas estándar



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
>>> import sys
>>> sys.path
['', 'C:\\Python27\\Lib\\idlelib', 'C:\\Windows\\system32\\python27.zip', 'C:\\P
ython27\\DLLs', 'C:\\Python27\\lib', 'C:\\Python27\\lib\\plat-win', 'C:\\Python2
7\\lib\\lib-tk', 'C:\\Python27', 'C:\\Python27\\lib\\site-packages']
>>> |
```

Ln: 267 Col: 4

# Introducción a Python

## Paquetes

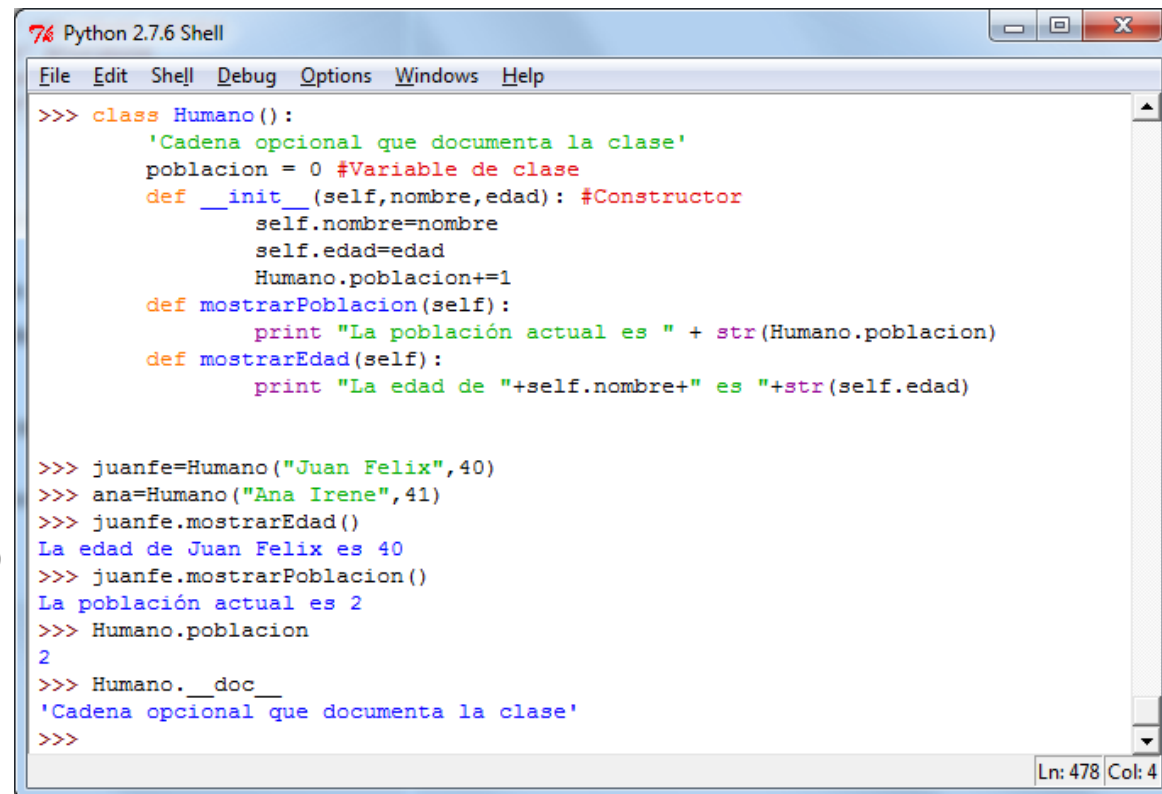
- Un paquete es un directorio en el que disponemos de varios módulos (archivos .py) y en el que incorporamos además un archivo llamado `__init__.py` (que puede estar simplemente vacío)
- Estos directorios suelen tener la inicial en **mayúscula** para distinguirlos de los módulos estándar (que están todos en minúsculas)
- Por ejemplo, podríamos agrupar en un paquete llamado “Matrix” varios módulos diseñados para realizar operaciones matriciales: Producto, Inversa, Determinante, ...
  - Para importar uno de estos módulos (Determinante) usaríamos indistintamente
    - `import Matrix.Determinante` (notación de punto)
    - `from Matrix import Determinante`
  - Pero también podríamos importarlos todos con **`from Matrix import *`** si en el archivo `__init__.py` insertamos la siguiente instrucción:
    - `__all__=["Producto","Inversa","Determinante"]`



# Introducción a Python

## Programación orientada a objetos

- Conceptos esenciales
  - Clase: Humano
  - Constructor: `__init__`
  - `self`
  - Objeto: `juanfe`
  - Variable de clase: `poblacion`
  - Variable de instancia: `edad`
  - Método: `mostrarEdad`  
Todos los métodos deben recibir al menos el argumento `self`



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help

>>> class Humano():
    'Cadena opcional que documenta la clase'
    poblacion = 0 #Variable de clase
    def __init__(self,nombre,edad): #Constructor
        self.nombre=nombre
        self.edad=edad
        Humano.poblacion+=1
    def mostrarPoblacion(self):
        print "La población actual es " + str(Humano.poblacion)
    def mostrarEdad(self):
        print "La edad de "+self.nombre+" es "+str(self.edad)

>>> juanfe=Humano("Juan Felix",40)
>>> ana=Humano("Ana Irene",41)
>>> juanfe.mostrarEdad()
La edad de Juan Felix es 40
>>> juanfe.mostrarPoblacion()
La población actual es 2
>>> Humano.poblacion
2
>>> Humano.__doc__
'Cadena opcional que documenta la clase'
>>>
```

Ln: 478 Col: 4

# Introducción a Python

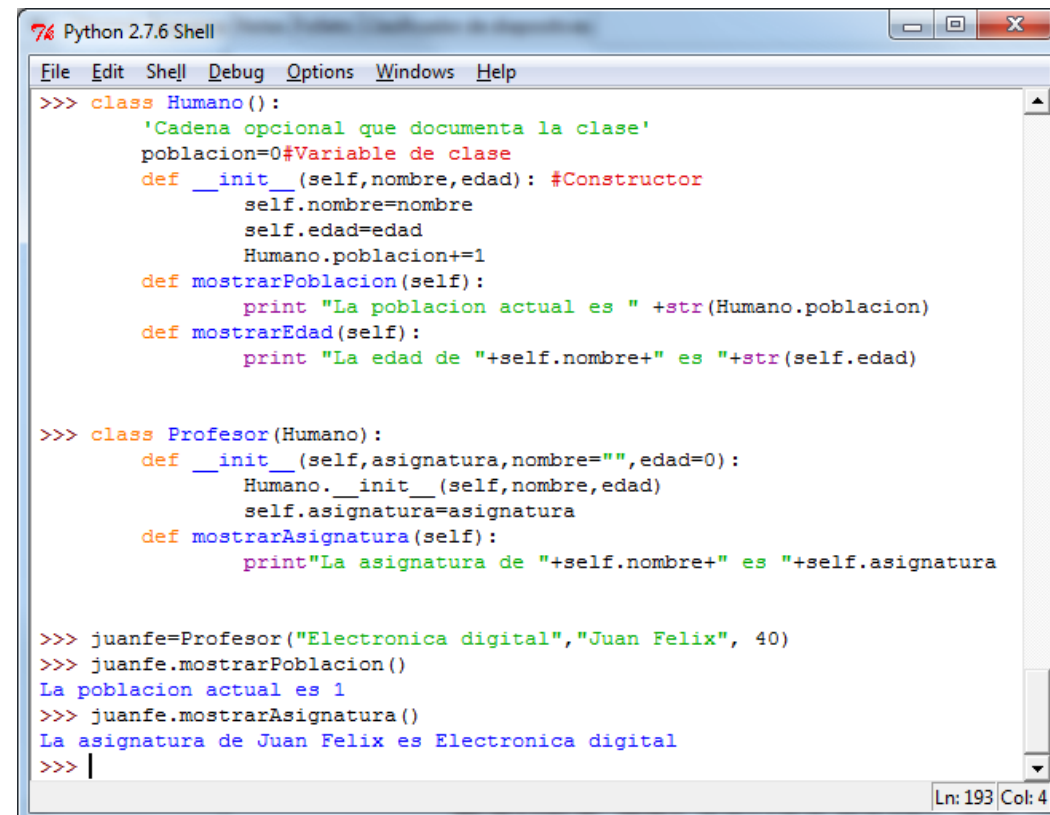
## Programación orientada a objetos

```
class Humano():  
    'Cadena opcional que documenta la clase'  
    poblacion=0#Variable de clase  
    def __init__(self,nombre,edad): #Constructor  
        self.nombre=nombre  
        self.edad=edad  
        Humano.poblacion+=1  
    def mostrarPoblacion(self):  
        print "La poblacion actual es " +str(Humano.poblacion)  
    def mostrarEdad(self):  
        print "La edad de "+self.nombre+" es "+str(self.edad)
```

# Introducción a Python

## Programación orientada a objetos: herencia

- Una clase puede heredar las variables y métodos de otras, ampliándolas con otras variables y/o métodos.
  - Basta con indicar el nombre de la clase de la que hereda dentro de los paréntesis que hay a la derecha de su nombre
    - `class Profesor(Humano):`  
#Profesor hereda de Humano
  - Dentro del constructor de la clase que hereda deberemos llamar al constructor de la clase de la que hereda:
    - `Humano.__init__(self,...)`



```
>>> class Humano():
    'Cadena opcional que documenta la clase'
    poblacion=0#Variable de clase
    def __init__(self,nombre,edad): #Constructor
        self.nombre=nombre
        self.edad=edad
        Humano.poblacion+=1
    def mostrarPoblacion(self):
        print "La poblacion actual es " +str(Humano.poblacion)
    def mostrarEdad(self):
        print "La edad de "+self.nombre+" es "+str(self.edad)

>>> class Profesor(Humano):
    def __init__(self,asignatura,nombre="",edad=0):
        Humano.__init__(self,nombre,edad)
        self.asignatura=asignatura
    def mostrarAsignatura(self):
        print"La asignatura de "+self.nombre+" es "+self.asignatura

>>> juanfe=Profesor("Electronica digital","Juan Felix", 40)
>>> juanfe.mostrarPoblacion()
La poblacion actual es 1
>>> juanfe.mostrarAsignatura()
La asignatura de Juan Felix es Electronica digital
>>> |
```

Ln: 193 Col: 4

# Introducción a Python

## Programación orientada a objetos: herencia

```
class Profesor(Humano):  
    def __init__(self, asignatura, nombre="", edad=0):  
        Humano.__init__(self, nombre, edad)  
        self.asignatura = asignatura  
    def mostrarAsignatura(self):  
        print("La asignatura de "+self.nombre+" es "+self.asignatura)
```

# Aplicaciones (módulos) en OpenERP

- Existe un documento en la documentación de Odoo llamado Technical Memento en el que se recoge información actualizada y resumida sobre la programación de módulos en OpenERP.
- En el momento de redactar este texto, este documento está disponible en:
  - <http://doc.openerp.com/memento>

# Aplicaciones (módulos) en OpenERP

- Una App (módulo) de OpenERP es sencillamente un paquete de Python, es decir un directorio.
- Además del archivo `__init__.py`, deberemos crear otro llamado `__openerp__.py`, que contiene un diccionario con una descripción de metadatos del módulo (nombre, versión, autor, ...). Este archivo se suele llamar “manifiesto”.
- `__init__.py`
  - En este archivo indicaremos la información de importación de módulos:
    - `__all__=[xxx,yyy,zzz]`
    - `import nombre del modulo sin la extensión py`
- `__openerp__.py`: Contiene un diccionario de metadatos

## \_\_openerp\_\_.py

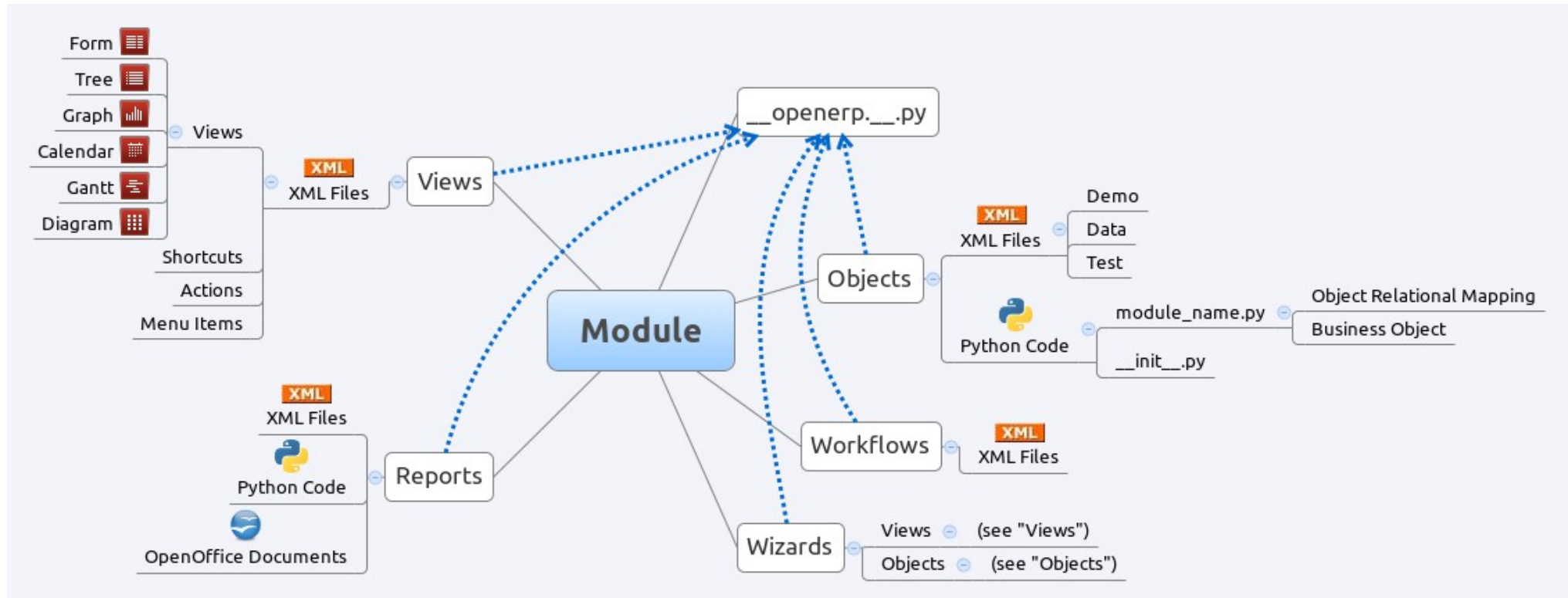
- **name:** Nombre del módulo en inglés
- **version:** Versión del módulo
- **summary:** Descripción breve del módulo
- **description:** Descripción completa del módulo
- **category:** Categoría a la que pertenece el módulo (ej. Tools o Account Charts)
- **author:** Nombre del autor del módulo
- **website:** URL del autor del módulo
- **license:** Por defecto AGPL-3
- **depends:** Tupla con los nombre de las apps de la que depende la que estamos desarrollando
  - Generalmente todos los módulos dependen del módulo base
- **data:** Tupla con los nombres de los archivos XML (vistas, datos iniciales, ...) del módulo
- **demo:** Tupla con los nombre de los archivos XML que incluyen los datos de ejemplo (se utilizan si el usuario ha creado la empresa con la opción Datos de ejemplo activada)
- **installable:** True or False. True para que se pueda instalar.
- **auto\_install:** True or False (por defecto False). El valor True haría que el módulo se instalase automáticamente si se detecta que ya están instalados todos los módulos de los que depende.

## Ejercicio

- Explorar los archivos `__init__.py` y `__openerp__.py` de varios de los módulos de la carpeta addons de OpenERP.
- Comprobamos que en los `__init__.py` existe un import para importar un módulo (archivo de python) con el mismo nombre que la propia app. Por ejemplo, en el `__init__.py` de la app CRM encontramos la línea **`import crm`** para importar el módulo `crm.py`.



# Estructura general de una app OpenERP



# Estructura general de una app OpenERP

## Carpetas y archivos

- addons/
  - visitas/
    - demo/ → Datos demostrativos para iniciar la app
    - i18n/ → Archivos de localización/traducción
    - report/ → Definición de los informes que añade la app
    - security/ → Definición de los grupos y privilegios que añade la app
    - view/ → Definición de las vistas, menús y acciones que añade la app
    - wizard/ → Definición de asistentes
    - workflow/ → Definición de flujos
    - \_\_init\_\_.py
    - \_\_openerp\_\_.py
    - miprimerapp.py

# ORM

## Object Relational Mapping ↔ `orm.Model`

- El servidor de OpenERP incluye una capa llamada ORM (Object Relational Mapping) que nos libera de tener que escribir código SQL para operar con la base de datos.
- ¿Cómo?
  - Todos los métodos de la capa ORM (crear un registro, buscar los registros que cumplen un criterio, leer registros, ...) están a nuestra disposición a través de una clase llamada **`orm.Model`**, de modo que sólo tenemos que hacer que nuestras clases hereden de ella para poder utilizarlos.
    - `class miprimerapp(orm.Model):`
  - Antes tendremos que importar el paquete `osv`, pero generalmente sólo se importan 2 de sus módulos en lugar del paquete completo:
    - **`from osv import orm, fields`**
  - En otras palabras, al crear una clase que herede de `orm.Model` estamos diciendo a OpenERP que nuestra clase va a poder operar sobre los registros de una tabla de la base de datos, pero...

## Pero... ¿de qué tabla?

### Atributos de orm.Model

- Mediante los atributos de orm.Model indicaremos si queremos crear una tabla nueva o aprovechar una de las ya existentes (por ejemplo, para crear una app que añada funcionalidades a otra app ya existente)
- Los principales atributos de orm.Model son:
  - **\_name**: el nombre de la tabla que queremos crear (o el de una tabla ya existente que queramos duplicar → Herencia de clase)
  - **\_columns**: diccionario con la descripción de los campos que queremos incluir en la tabla
  - **\_defaults**: valores predeterminados para los campos anteriores
  - **\_inherit**: el nombre del objeto que queremos ampliar en caso de que deseemos ampliar una tabla ya existente
  - **\_order**: nombre del campo que se usará para ordenar los registros (por defecto, se utiliza el campo id)

## Pero... ¿de qué tabla?

### Atributos de orm.Model

```
from osv import orm, fields

class miprimerapp(orm.Model)

    _name = 'visitas.visitas'
    _columns = {...}
    _defaults = {...}
```

- ¿Por qué hemos puesto como nombre visitas.visitas?
  - El punto será sustituido por un \_ en el nombre de la tabla.
  - Es una buena práctica porque si en el futuro necesitamos otras tablas a este módulo podremos llamarlas visitas.otratabla, y así aparecerán todas agrupadas en pgAdminIII.

# Tipos de campos en la capa ORM

- Básicamente existen 3 tipos de campos:
  - **simples** o clásicos: campos que contienen un dato concreto (un entero, una cadena de caracteres, una imagen, ...)
  - **relacionales**: campos que representan una relación con otras tablas (one2many, many2one, o many2many)
  - **funcionales**: estos campos no se almacenan en la base de datos sino que su valor se establece en tiempo de ejecución mediante funciones python o leyéndolos de otros campos.
- Cada campo se define como un par nombre:valor del diccionario `_columns`
  - `_columns = {'apellido': fields.char('Apellido', size=50, required=True)}`

## Atributos comunes de todos los tipos de campos

- Todos los campos comparten los siguientes atributos comunes (entre otros):
  - **string** → La etiqueta del campo en la tabla
  - **required** → True si queremos que no pueda quedar en blanco
  - **select** → True si queremos que la tabla se indexe por esta columna
  - **readonly** → True si queremos que el valor del campo no se pueda cambiar
  - **help** → Un texto de ayuda que aparezca como tooltip al colocar el puntero del ratón sobre el campo

# Principales tipos de campos simples

- **boolean**(string, ...)
  - 'archivado' : fields.boolean('Archivado')
- **integer**(string, ...)
  - 'edad' : fields.integer('Edad')
- **date**(string, ...)
  - 'nacimiento' : fields.date('Fecha denacimiento')
- **char**(string,size, ...)
  - 'apellido' : fields.char('Apellido', size=50, required=True)
- **float**(string,digits, ...) → digits es una tupla con el formato (enteros,decimales)
  - 'altura' : fields.float('Altura',digits=(3,1))
- **selection**(values, string...) → values es una lista de tuplas valor,nombre
  - 'sexo' : fields.selection([('h','Hombre'),('m','Mujer')],'Sexo')
- **binary**(string, ...)
  - 'fotografia' : fields.binary('Fotografia')

¡Ojo!  
En selection los  
valores van  
delante del  
string



# Ejercicio

## Nuestra primera app

- Crear en la carpeta addons una subcarpeta llamada visitas y dentro de ella:
  - el archivo `__init__.py`
  - el archivo `__openerp__.py`
  - el archivo `visitas.py` que cree una tabla llamada `visitas_visitas` con un único campo
    - nombre
      - `string = Nombre`
      - Tipo: `char`
      - `size = 50`
      - `required = True`

## Ejercicio

### Nuestra primera app: `__init__.py`

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

```
import visitas
```

## Ejercicio

### Nuestra primera app: `__openerp__.py`

```
# -*- coding: utf-8 -*-
{
    'name': 'Visitas',
    'version': '1.0',
    'category': 'Other',
    'summary': 'Este módulo es un control de accesos sencillo',
    'description': '''
Anote el nombre y las horas de entrada y salida
de los visitantes de su empresa
''',
    'author': 'Juan Felix Mateos',
    'depends': ['base'],
    'installable': True
}
```

# Ejercicio

## Nuestra primera app: visitas.py

```
# -*- coding: utf-8 -*-  
from osv import orm, fields  
class visitas(orm.Model):  
    _name = 'visitas.visitas'  
    _columns = {  
        'nombre' : fields.char(string='Nombre', size=50, required=True)  
    }
```

# Ejercicio

## Nuestra primera app: instalar

- Configuración>Módulos>Actualizar lista de módulos
- En Módulos locales, localizar la aplicación visitas en la categoría Otros (agrupar la vista de lista de módulos locales por Categoría).
- Instalar la aplicación
- ¿Qué ha pasado?
  - Comprobar con pgAdminIII que se ha creado la tabla visitas\_visitas
  - ¿Cuántos campos tiene?
    - 5 además del que hemos creado nosotros
  - ¿Por qué?
    - Porque nuestra clase, al heredar de orm.Model ha adquirido las columnas que impone orm.Model
      - id: id único de cada registro (clave primaria)
      - create\_uid: id del usuario que crea el registro
      - create\_date
      - write\_date
      - write\_uid: id del último usuario que ha modificado el registro

# Ejercicio

## Nuestra primera app

- Modificar el archivo visitas.py para incluir además los siguientes campos
  - sexo: selection (con los valores h y m para hombre y mujer)
  - edad: integer
  - altura: float (con 3 enteros y 1 decimal)
  - fotografia: binary
  - entrada: date
  - salida: date
- Para que OpenERP detecte los cambios tenemos que reiniciar el servidor de OpenERP (services.msc) y, después, actualizar el módulo (botón **Update** en la ficha del módulo)
- Update vs Upgrade: Update ejecuta el código del módulo actual sobre la base de datos, mientras que Upgrade reemplaza el módulo actual con la versión más reciente disponible en el servidor.

# Ejercicio

## Nuestra primera app: visitas.py

```
# -*- coding: utf-8 -*-
from osv import orm, fields
class visitas(orm.Model):
    _name = 'visitas.visitas'
    _columns = {
        'nombre' : fields.char('Nombre', size=50, required=True),
        'sexo' : fields.selection([('h','Hombre'),
('m','Mujer')], 'Sexo'),
        'edad' : fields.integer('Edad'),
        'altura' : fields.float('Altura', digits=(3,1)),
        'fotografia' : fields.binary('Fotografia'),
        'entrada' : fields.date('Entrada'),
        'salida' : fields.date('Salida')
    }
```

## Modelo ↔ Vista ↔ Controlador

- La capa superior de OpenERP (la capa de presentación) utiliza el patrón MVC (Modelo, Vista, Controlador).
- El modelo es precisamente lo que acabamos de crear con el archivo `visitas.py` al heredar de `orm.Model`.
  - Cualquier operación que deseemos realizar sobre los datos la codificaremos como un método de la clase `visitas` (o como un método de otra clase que herede de ella).
- El controlador está conformado por acciones, que el usuario puede ejecutar mediante menús o botones.
- La vista es la interfaz en la que se presentan los datos y se ofrecen los menús/botones que lanzan las acciones.



## Modelo ↔ Vista ↔ Controlador

- Tanto las acciones, como las vistas e incluso los menús se codifican en OpenERP mediante archivos XML.
- Para vincular estos archivos XML al módulo se utiliza el atributo data de la colección del manifiesto (`__openerp__.py`).
- Por ejemplo, configurar nuestro manifiesto para que utilice un archivo xml llamado **visitas\_view.xml**

## Modelo ↔ Vista ↔ Controlador

- Por ejemplo, configurar nuestro manifiesto para que utilice un archivo xml llamado **visitas\_view.xml**

```
# -*- coding: utf-8 -*-
{
    'name': 'Visitas',
    'version': '3.0',
    'category': 'Other',
    'summary': 'Este módulo es un control de accesos sencillo',
    'description': '''
Anote el nombre y las horas de entrada y salida
de los visitantes de su empresa
''',
    'author': 'Juan Felix Mateos',
    'depends': ['base'],
    'installable': True,
    'data': ['visitas_view.xml']
}
```

# Estructura general de un archivo XML de OpenERP

- Cada archivo XML que queramos usar en un módulo OpenERP deberá tener la siguiente estructura
  - Los registros se codifican con el elemento **<record>** y pueden ser vistas, acciones o menús.

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

    <!-- Aquí irán los registros -->

  </data>
</openerp>
```

## Registros de vistas

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

- Las vistas pueden ser de muchos tipos
  - Formularios
  - Árbol
  - Gráfico
  - Búsqueda
  - Calendario
  - Kanban
  - Gantt
- No obstante, los más utilizados son Formularios y Árbol
- Para crear una vista utilizaremos un elemento **record** con el modelo `ir.ur.view`
  - `<record model="ir.ui.view" id="nombre_de_la_vista">`

# Contenido de los registros de vistas

## Campos

- Dentro del record de la vista deberemos crear varios elementos field como:
  - `<field name='name'>visitas.tree</field>`
    - Indica el nombre de la vista
  - `<field name='model'>visitas.visitas</field>`
    - Indica sobre qué model va a actuar la vista (los datos de qué modelo va a mostrar o modificar).
  - `<field name='type'>tree</field>`
    - Indica de qué tipo es la vista (tree, form, kanban, ...)
  - `<field name='arch' type='xml'>`
    - Dentro de este elemento se codificará el contenido de la vista

# Ejercicio

## Iniciar el archivo visitas\_view.xml

- Crear el archivo visitas\_view.xml e iniciar en él la creación de una vista de tipo árbol

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

  <record model="ir.ui.view" id="visitas_view_tree">
    <field name="name">visitas.tree</field>
    <field name="model">visitas.visitas</field>
    <field name="type">tree</field>
    <field name="arch" type="xml">
      <!-- Aquí irá el contenido de la vista -->
    </field>
  </record>

</data>
</openerp>
```

## El contenido de las vistas

- El contenido de la vista se codificará dentro de un elemento anidado en el field de tipo arch:
  - `<tree>`
  - `<form>`
  - `<kanban>`
  - ...
- Estos elementos admiten un atributo string para el rótulo de la vista
  - `<tree string='Listado de visitas'>`
- Dentro del elemento anterior se anidarán distintos elementos para componer la vista:
  - `<field>`
  - `<button>`
  - `<group>`
  - ...

# Ejercicio

## Iniciar el contenido de la vista de árbol

- Crear el elemento `<tree>` dentro de la vista que iniciamos anteriormente

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

    <record model="ir.ui.view" id="visitas_view_tree">
      <field name="name">visitas.tree</field>
      <field name="model">visitas.visitas</field>
      <field name="type">tree</field>
      <field name="arch" type="xml">
        <tree string="Listado de visitas">
          <!-- Aquí irá el contenido de la vista -->
        </tree>
      </field>
    </record>

  </data>
</openerp>
```



## Mostrar campos del modelo en una vista

- Para mostrar un campo de un modelo dentro de una vista simplemente tendremos que crear un elemento `<field>` cuyo atributo `name` coincida con el nombre del campo (en la base de datos).
  - `<field name="Nombre"/>`

## Ejercicio

### Insertar en la vista de árbol todos los campos

- Insertar dentro del elemento `<tree>` un elemento `field` para cada uno de los campos del modelo (excepto la fotografía, que no se puede mostrar bien en este tipo de vista).

```
...  
<field name="arch" type="xml">  
  <tree string="Listado de visitas">  
    <field name="nombre"/>  
    <field name="sexo"/>  
    <field name="edad"/>  
    <field name="altura"/>  
    <field name="entrada"/>  
    <field name="salida"/>  
  </tree>  
</field>  
...
```

## Ejercicio

### Crear una segunda vista de formulario

- En el mismo archivo XML, crear una segunda vista de tipo formulario, pero que esta vez sí incluya el campo de fotografía
  - Lo único que cambia respecto a la de árbol es
    - El `<field name="type">` deberá ser `form` en lugar de `tree`.
    - En lugar de usar el elemento `<tree>` usaremos el elemento `<form>`

# Ejercicio

## Crear una segunda vista de formulario

```
<record model="ir.ui.view" id="visitas_view_form">
  <field name="name">visitas.form</field>
  <field name="model">visitas.visitas</field>
  <field name="type">form</field>
  <field name="arch" type="xml">
    <form string="Listado de visitas">
      <field name="fotografia"/>
      <field name="nombre"/>
      <field name="sexo"/>
      <field name="edad"/>
      <field name="altura"/>
      <field name="entrada"/>
      <field name="salida"/>
    </form>
  </field>
</record>
```

# Acciones

- En un módulo podemos crear distintos tipos de acciones:
  - Abrir una vista
  - Imprimir un informe
  - Iniciar un asistente (Wizard)
  - ...
- Todas ellas, como ya comentamos anteriormente, se codifican con el elemento `<record>`, exactamente igual que las vistas.
- Lo único que cambia es que en lugar de usar el atributo `model` **ir.ui.view** de las vistas, utilizan el `model` **ir.actions.act\_window**

# Contenido de los registros de acciones

## Campos

- Dentro del record de la acción deberemos crear varios elementos field como:

- `<field name='name'>Visitas</field>`

- Indica el nombre de la acción

- `<field name="view_id" ref="visitas_view_tree"/>`

- Indica el id de la vista que queremos abrir

- `<field name='res_model'>visitas.visitas</field>`

- Indica sobre qué model va a actuar la vista que va a abrir la acción (los datos de qué modelo va a mostrar o modificar).

- `<field name='view_type'>form</field>`

- Puede adquirir los valores form (visualización (individual) y edición de registros) o tree (sólo visualización (en listado) de registros)

- `<field name='view_mode'>form,tree</field>`

- Si el view\_type es form, este campo nos permite indicar qué otros tipos de vistas están disponibles

¡Ojo! Su sintaxis es distinta a la de los demás porque usa ref

# Ejercicio

## Añadir al archivo XML una acción

- Añadir al archivo XML una acción que abra la vista de árbol visitas\_view\_tree

```
<record model="ir.actions.act_window" id="action_visitas">  
  <field name="name">Visitas</field>  
  <field name="view_id" ref="visitas_view_tree"/>  
  <field name="res_model">visitas.visitas</field>  
  <field name="view_type">form</field>  
  <field name="view_mode">tree,form</field>  
</record>
```

# Menús

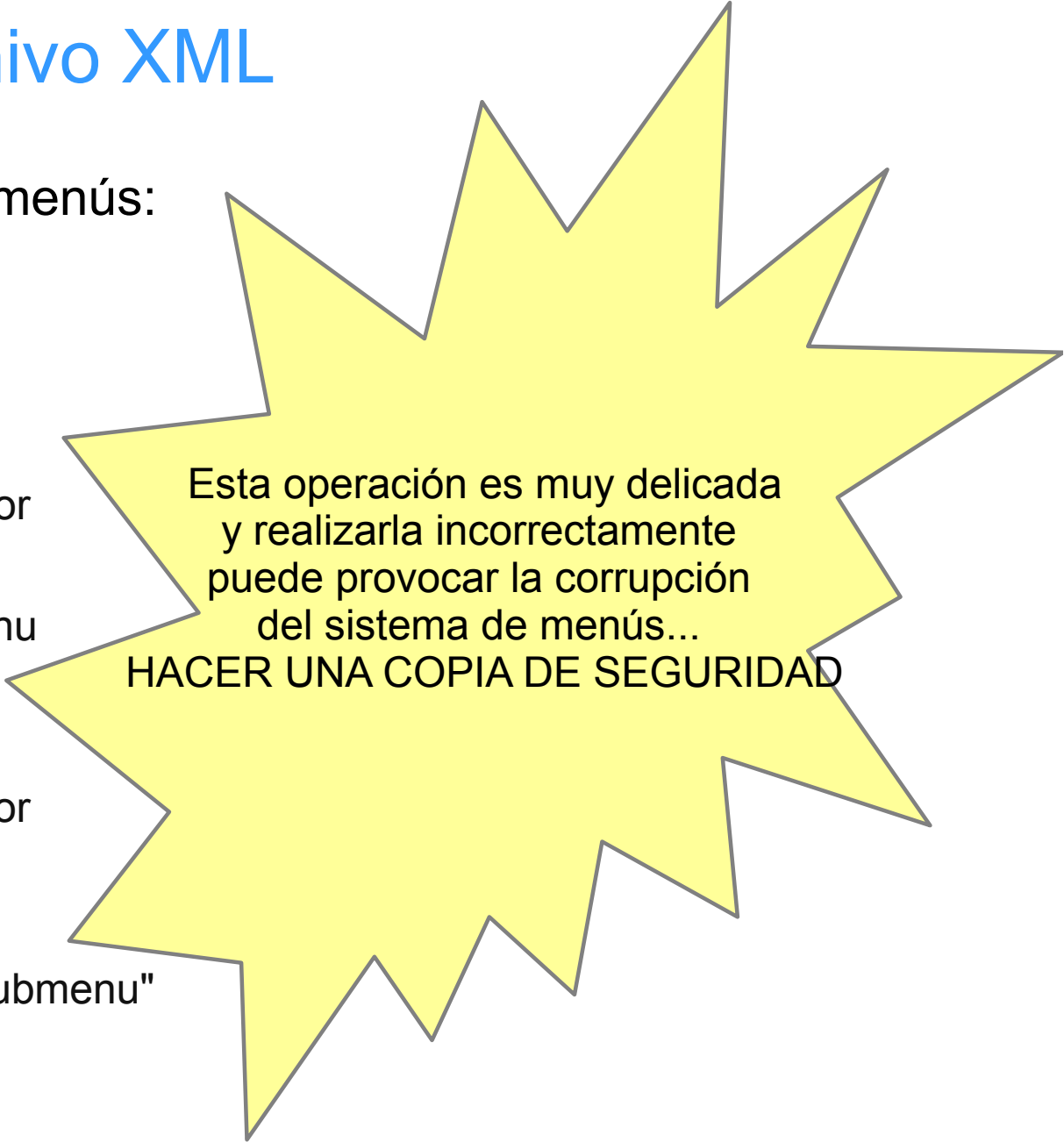
- Los menús pueden tener asociada una acción, o simplemente actuar como rótulos para contener otros menús (menús anidados).
- En OpenERP 7 hay un menú principal que se muestra en la zona superior y un menú secundario anidado en él que se muestra en el lateral izquierdo.
- En versiones anteriores de OpenERP los menús también se creaban a través de elementos `<record>`, que en ese caso utilizaban el model `ir.model.data`
- Sin embargo, en OpenERP 7 y posteriores se recomienda reemplazarlo por el elemento `<menuitem>` con los siguientes atributos:
  - `id`: identificador único
  - `parent`: si se trata de un submenú, para indicar el id del menú del que depende
  - `sequence`: 10, 20, 30... para ordenar el menú entre sus “hermanos”
  - `action`: id de la acción que ejecuta el menú



# Ejercicio

## Añadir menús al archivo XML

- Crear en el archivo XML 3 menús:
  - Menú 1:
    - name: Visitas
    - id: visitas\_principal
    - sequence: 60
  - Menú 2: Anidado en el anterior
    - name: Visitas
    - id: visitas\_grupo\_submenu
    - parent: visitas\_principal
    - sequence: 1
  - Menú 3: Anidado en el anterior
    - name: Visitas
    - id="visitas\_submenu"
    - parent="visitas\_grupo\_submenu"
    - action="action\_visitas"
    - sequence="1"



Esta operación es muy delicada  
y realizarla incorrectamente  
puede provocar la corrupción  
del sistema de menús...  
**HACER UNA COPIA DE SEGURIDAD**

# Ejercicio

## Añadir menús al archivo XML

```
<menuitem name="Visitas"
  id="visitas_principal"
  sequence="60"/>
<menuitem name="Visitas"
  id="visitas_grupo_submenu"
  parent="visitas_principal"
  sequence="1" />
<menuitem name="Visitas"
  id="visitas_submenu"
  parent="visitas_grupo_submenu"
  action="action_visitas"
  sequence="1" />
```

# Ejercicio

## Probar el módulo

- Reiniciar el servidor de OpenERP
- Actualizar la lista de módulos (Configuración>Actualizar lista de módulos)
- Actualizar el módulo (con el botón **Update** de la ficha del módulo)
- Crear un registro de entrada en el módulo

Visitas / Nuevo

**Guardar** o Descartar



Fotografia

Seleccionar Guardar como Limpiar

Nombre

Sexo

Edad

Altura

Entrada

Salida

# Ejercicio

## Mejorar la presentación de la fotografía

- Para mejorar el aspecto de la presentación de la fotografía en el campo de formulario podemos añadir al elemento `<field>` el atributo `widget="image"`
  - `<field name="fotografia" widget="image" options="{ 'preview_image': 'image_medium', 'size': [90, 90]}"/>`

Visitas / visitas.visitas,1

Editar Crear Más ▾

Fotografia		Nombre	Juan Félix Mateos
Sexo	Hombre	Edad	41
Altura	185,5	Entrada	03/07/2014
Salida	02/07/2014		

# Ejercicio

## Crear un campo relacional

- Vamos a sustituir el campo nombre de nuestro modelo por un campo relacional many2one vinculado a la tabla res.partner
- En el archivo visitas.py, sustituir el campo nombre por el siguiente

```
'nombre_id' : fields.many2one('res.partner', 'Nombre'),
```

- En el archivo visitas\_view.xml cambiar el campo del nombre en las 2 vistas por:

```
<field name="nombre_id"/>
```

## Ejercicio

### Crear un campo vinculado a otro

- Existe un tipo de campo especial (`fields.related`) que nos permite vincular su contenido a otro campo de otra tabla.
- Queremos vincular nuestro campo de Fotografía al avatar del usuario que se haya elegido en Nombre.
- `fields.related` requiere recibir como primer argumento una lista del path necesario para alcanzar al campo de la otra tabla. En nuestro caso ese path será `'nombre_id','image'` porque `'nombre_id'` contiene el id de la tabla `partner_id`.
- Otros 2 argumentos importantes de este tipo de campos son:
  - `type` que en nuestro caso será `type='binary'`
  - `relation` que es la tabla a la que queremos vincular, que en nuestro caso será `relatio='res.partner'`

# Ejercicio

## Crear un campo vinculado a otro

- En el archivo `visitas.py`, cambiar el campo `fotografía` por:

```
'fotografia' :  
fields.related('nombre_id', 'image', type='binary', relation='res.par  
tner', string='Fotografía'),
```

- En el archivo de la vista de formulario, cambiar el campo de la fotografía por

```
<field name="fotografia" widget="image" readonly="1"/>
```

## Asociar un icono a un módulo

- Para asociar un icono a un módulo tendremos que:
  - Crear una imagen PNG de 64x64 píxeles
  - Almacenar esta imagen con el nombre icon.png en la carpeta visitas\static\src\img de nuestro módulo
  - En el manifiesto, añadir al diccionario el par 'application': True

```
# -*- coding: utf-8 -*-
{
    'name': 'Visitas',
    'version': '3.0',
    'category': 'Other',
    'summary': 'Este módulo es un control de accesos sencillo',
    'description': '''
Anote el nombre y las horas de entrada y salida
de los visitantes de su empresa
''',
    'author': 'Juan Felix Mateos',
    'depends': ['base'],
    'installable': True,
    'data': ['visitas_view.xml'],
    'application': True
}
```