Formación técnica – Soluciones

Versión 7.0

Contenido

6. Relaciones entre Objetos	
6.1. Campos relacionales.	
7. Herencia	
7.1. Mecanismos de herencia	
7.2. Herencia de vista	
Domains (Dominios)	
8. Métodos	
8.1. Funciones en campos	
8.2. Onchange	
9. Vistas Avanzadas	
9.1. Lista y Tree	
9.2. Calendario	
9.3. Buscar vistas	
9.4. Graficos	
9.5. Tablero Kanban	
10. Flujos de Trabajo	
11. Seguridad	
11.1. Grupos basados en mecanismos de control de acceso	
11.2. Derecho de acceso	
11.3. Nomas de registro	
12. Wizards	
12.1. Objetos Wizards	
12.2. Ejecución de Wizard	
13. Internacionalización	
14. Reportes	
14.1. Impresión de reportes	
Expresiones utilizadas en las plantillas de reportes OpenERP	
Reportes RML	
Reportes WebKit	
14.2. DashBoards	
15. WebServices	
15.1. Biblioteca XML-RPC	
15.2. Biblioteca OpenERP Client	

Basado en un ejemplo real, este manual contiene: la elaboración de un módulo de OpenERP con sus respectivas interfaz, Vistas, Los informes, flujos de trabajo, los aspectos de seguridad, Wizards, WebServices, internacionalización, Rapid Application Development (RAD) y optimización del rendimiento.

1. Acerca de OpenERP

1.1. Punto de vista funcional

- Mostrar OpenERP
- Mostrar principales conceptos funcionales en una db demostración
  - Menús
  - Vistas
    - ✓ Lista + Buscar
    - ✓ Forma
    - ✓ Calendario, Gráfico, etc
  - Módulos
  - Usuarios
  - ¿Empresas?
  - Objetos principales (de una manera funcional!)
    - ✓ Partner (socio)
    - ✓ Producto

1.2. Punto de vista técnico

- **Estructura general, el esquema**
  - Servidor - Cliente - Cliente Web
  - Comunicación: Net-RPC, XML-RPC
- **Framework – ORM**
  - ¿Qué es un ORM? - Objeto, relaciones entre objetos
  - OpenERP ORM particularidades
    - ✓ 1 instancia de un objeto de la lógica no es el 1 de registro en la base de datos
    - ✓ 1 instancia de un objeto de la lógica es 1 mesa en el DB
    - ✓ Breve resumen del objeto OSV, métodos ORM

1.3. Instalación en una máquina Unix con fines de desarrollo

Bazaar

- Versiones: principio general
- Sistema de control de versiones centralizado vs sistema de control de versiones distribuido
- Comandos BZR principales

Descarga e instalación de OpenERP

La instalación de la versión OERP en Linux:

- **Introducción a bzr primero**
  - Instalación Bzr
  - Comandos principales
  - Detalles sobre cómo van a utilizarse
- **Descarga de Server, Client, Complementos**
  - Observaciones sobre extra-addons, comunidad-addons

- **Instalación de Dependencias**
  - PostgreSQL: Instalación, añadir nuevos usuarios
  - Openerp configuración: cómo crear OpenERP-serverrc
  - Python dependencias

2. Introducción ha Python

Python es un lenguaje de indotación sensible imperativo / orientado a objetos, utilizando características de declaración/lenguajes funcionales (estas características están fuera del alcance de la formación actual). Para poder acceder al tutorial de python oficial ingresa a: http://docs.python.org/tutorial/index.html

2.1. Interpretador Python

Al escribir un carácter al final de un archivo (Control-D en Unix, Control-Z en Windows) en el indicador principal causa que el intérprete de un código de salida de cero. Si eso no funciona, puede salir del intérprete tecleando la siguiente comando: *quit ()*.

En el modo interactivo que solicita el comando siguiente con el mensaje principal, usualmente tres signos mayor que (>>>), Para las líneas de continuación se le pedirá con el indicador secundario, por defecto tres puntos (...).

Se necesitan líneas de continuación al entrar en una construcción de varias líneas. A modo de ejemplo, echar un vistazo a esta sentencia if:

```
>>> el_mundo_es_plano = 1
>>> if el_mundo_es_plano:
...     print "Ten cuidado de no caerte!"
...
...
Ten cuidado de no caerte!
```

(Ejemplo de un if)

En los sistemas Unix tipo BSD, los scripts de Python se pueden hacer directamente ejecutable, como los scripts de shell, poniendo la línea.

```
#! / usr / bin / env python
```

(Suponiendo que el intérprete está en el PATH del usuario) al comienzo de la secuencia de comandos y dar al archivo un modo ejecutable.

2.2. Números

El intérprete actúa como una simple calculadora: se puede escribir una expresión en ella y se escribe el valor. La sintaxis es sencilla: los operadores *+, -, ** y */* funcionan igual que en la mayoría de otros idiomas (por ejemplo, Pascal o C); paréntesis se pueden utilizar para la agrupación. Por ejemplo:

```
>>> 2+2
4
>>> # Esto es un comentario
... 2+2
4
>>> 2+2 # Y esto es un comentario en la misma linea de un codigo
4
>>> (50-5*6)/4
5
>>> # Division de enteros
... 7/3
2
>>> 7/-3
-3
```

El signo igual ('=') se utiliza para asignar un valor a una variable. Posteriormente, ningún resultado se muestra antes del siguiente modo interactivo:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Un valor se puede asignar a varias variables simultáneamente:

```
>>> x = y = z = 0 # X, Y, Z valen Zero
>>> x
0
>>> y
0
>>> z
0
```

Las variables deben ser "definidas" (asignarles un valor) antes de que puedan ser utilizadas, o se producirá un error:

```
>>> #  Acceder a una variable no definida
... n
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Hay compatibilidad total para el punto flotante (float); realizar operaciones mixtas, convirtiendo un operación entera (interger) a una flotante (float):

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Tambien pueden realizarse funciones de conversión de flotantes a enteros (float (), int () y long ()).

2.3. Cadenas

Además de números, Python también puede manipular cadenas, que se puede expresar de varias maneras. Ellas pueden ir entre comillas simples o dobles:

```
>>> 'spam eggs' #Comillas Simples
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't" #Comillas Dobles
"doesn't"
>>> '"Yes," he said.' #Comillas Simples y Dobles
```

Las cadenas pueden ser concatenadas (pegadas) con el operador +, y se repetirse con el operador *:

```
>>> Mundo = 'Ayuda' + 'A'
>>> Mundo
'AyudaA'
>>> '<' + word*5 + '>'
'<AyudaAAyudaAAyudaAAyudaAAyudaA>'
```

Dos cadenas juntas una a la otra se concatenan automáticamente, la primera línea de arriba también podría haber sido escrita palabra = 'Ayuda' 'A', esto sólo funciona con dos literales, no con expresiones de cadena arbitrarias:

```
>>> 'computa' 'dora' # <- Expresion valida
'computadora'
>>> 'computa'.strip() + 'dora' # <- Expresion valida
'computadora'
>>> 'computa'.strip() 'dora' # <- Expresion no valida
```

Las cadenas pueden ser subíndices (indexado); como en C, el primer carácter de una cadena tiene el (índice) 0. No hay ningún tipo de carácter independiente, un personaje es simplemente una cadena de longitud uno. Al igual que en el Icono, subcadenas se pueden especificar con la notación de corte: dos índices separados por dos puntos.

```
>>> Mundo[4]
'a'
>>> Mundo[0:2]
'Au'
>>> Mundo[2:4]
'ua'
```

Los índices pueden ser números negativos, para empezar a contar desde la derecha. Por ejemplo:

```
>>> Mundo[-1] # El ultimo caracter de la cadena
'A'
>>> Mundo[-2] # El Penultimo caracter de la cadena
'a'
>>> Mundo[-2:] # Los dos últimos caracteres
'aA'
>>> Mundo[:-2] # Todos, excepto los dos últimos caracteres de la cadena
'Ayud'
```

NOTA: Pero tenga en cuenta que -0 es lo mismo que 0, por lo que no se cuenta desde la derecha.

```
>>> Mundo[-0] # (desde -0 es igual a 0)
'A'
```

Cadenas y objetos de código unico (unicode) tienen una única función de operación: el operador **%** (módulo). Esto también es conocido como el formato de cadenas u operador de interpolación. Teniendo en cuenta los valores de **%** de formato (donde formato es una cadena o un objeto Unicode), especificaciones de conversión**%** en formato se sustituyen por cero o más elementos de los valores.

Si el formato requiere un solo argumento, los valores pueden ser un único objeto no tupla. De lo contrario, los valores deben ser una tupla con exactitud el número de elementos especificados por la cadena de formato, o un solo objeto de asignación (por ejemplo, un diccionario). Los posibles valores son:

- **'%d'** Firmado decimal entero.
- **'%e'** Flotando formato exponencial punto (en minúsculas).
- **'%E'** Flotando formato exponencial punto (en mayúsculas).
- **'%f'** Formato decimal de coma flotante.
- **'%F'** Formato decimal de coma flotante.
- **'%c'** Carácter individual (acepta cadena de caracteres entero o individual).
- **'%s'** String (convierte cualquier objeto de Python usando str ()).

He aquí un ejemplo.

```
>>> import math
>>> print 'El valor de PI es aproximadamente %5.3f.' % math.pi
El valor de PI es aproximadamente 3.142.
```

2.4. If

Tal vez el tipo más conocido de declaración es la instrucción if. Por ejemplo:

```
>>> x = int(raw_input("Por favor introduzca un número entero: "))
Por favor introduzca un número entero: 42
>>> if x < 0:
...     x = 0
...     print 'Negativo cambiado a cero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Simple'
... else:
...     print 'Mas'
...
Mas
```

Puede haber cero o más partes elif y la parte else es opcional. La palabra clave 'elif' es la abreviatura de 'else if'.

2.5. For

La declaración en Python difiere un poco de lo que puede estar acostumbrado en C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal), o dando al usuario la capacidad de definir tanto la etapa de iteración y condiciones de detención (como C), la instrucción for en Python itera sobre los elementos de cualquier secuencia (una lista o una cadena), en el orden en que aparecen en la secuencia. Por ejemplo (sin juego de palabras):

```
>>> # Medir algunas cadenas:
... a = ['gato', 'ventana', 'carro']
>>> for x in a:
... print x, len(x)
...
gato 4
ventana 7
carro 5
```

No es seguro modificar la secuencia que se repiten a lo largo del ciclo (esto sólo puede suceder por tipos secuenciales mutables, como las listas).

2.6. Listas

Python conoce una serie de tipos de datos compuestos, que se utilizan para agrupar otros valores. El más versátil es la lista, que se puede escribir como una lista de valores separados por comas (material) entre corchetes. Nombrar el material no es necesario que todos tienen el mismo tipo.

```
>>> a = ['carro', 'casa', 100, 1234]
>>> a
['carro', 'casa', 100, 1234]
```

Al igual que los índices de cadenas, listas índices comienzan en 0, y las listas se pueden cortar, concatenar y así sucesivamente.
Diferencia de las cadenas, que son inmutables, es posible cambiar los elementos individuales de una lista.

```
>>> a
['carro', 'casa', 100, 1234]
>>> a [2] = a [2] + 23
['carro', 'casa', 123, 1234]
```

Asignación por partes también es posible, y esto puede incluso cambiar el tamaño de la lista o borrar por completo:

```
>>> # Reemplace algunos items
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Quiramos algunos:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insertamos algunos:
... a[1:1] = ['carro', 'casa']
>>> a
[123, 'carro', 'casa', 1234]
>>> #Insertar (una copia) al principio
>>> a[:0] = a
>>> a
[123, 'carro', 'casa', 1234, 123, 'carro', 'casa', 1234]
>>> # Borrar la lista: reemplazar todos los elementos con una lista vacía
>>> a[:] = []
>>> a
[]
```

La función len () también se aplica a las listas:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

list.append (x) Añade un elemento al final de la lista, equivalente al [len (a):] = [x].
list.Count (x) Devuelve el número de veces que x aparece en la lista.
list.Sort () Ordene los elementos de la lista, en su lugar.
list.reverse () Invertir la lista de elementos, que se colocan desde la última a la primera.

Compresion de listas

Las comprensión de listas proporcionan una manera concisa de crear listas sin tener que recurrir al uso de map(), filter() y / o lambda. La definición de lista resultante tiende a menudo a ser más claro que las listas construidas con esas construcciones. Cada lista por comprensión consiste en una expresión seguida de una cláusula, a continuación, cero o más a favor o en caso de cláusulas.

El resultado será una lista, como resultado de la evaluación de la expresión en el contexto del *for* y *if* que le siguen. Si la expresión sería evaluar a una tupla, se debe colocar entre paréntesis.

```
>>> freshfruit = [' banana', ' frambuesa ', ' granadina ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'frambuesa', 'granadina']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3] #Si los valores de x son mayorores a 3 multiplique el valor
[12, 18]
>>> [3*x for x in vec if x < 2] #Si los valores de x son mayorores a 3 multiplique el valor
[]
>>> [[x,x**2] for x in vec] #Utilizacion de corhochetes asociacion de fuciones
[[2, 4], [4, 16], [6, 36]]

>>> [x, x**2 for x in vec] # error - paréntesis necesarios para tuplas
File "<stdin>", line 1, in ?
[x, x**2 for x in vec]
        ^
SyntaxError: invalid syntax
```

(ejemplo 1 copresion de listas)

```
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2] #Multiplicamos los valores de vec1 con cada uno de los valores de vec2
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2] #Sumamos los valores de vec1 con cada uno de los valores de vec2
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))] # Multiplicamos los atributos del vec1 con el vec2 en base a su posicion
[8, 12, -54]
```

(ejemplo 2 copresion de listas)

2.7. Compresion de listas

Podemos crear una función que escriba la serie de Fibonacci hasta un límite arbitrario. Es fácil de escribir una función que devuelve una lista de los números de la serie de Fibonacci, en lugar de imprimirlo.

```
>>> def fib2(n): # retorno serie de Fibonacci hasta n
... """Devuelve una lista que contiene la serie de Fibonacci hasta n"""
... result = []
... a, b = 0, 1
... while a < n:
... result.append(a)
... a, b = b, a+b
... return result
...
>>> f100 = fib2(100) #Llamamos
>>> f100 # Escribir el resultado
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Parámetros por defecto

La forma más útil es especificar un valor predeterminado para uno o más argumentos. Esto crea una función que puede ser llamada con menos argumentos de lo que se define para permitir. Por ejemplo:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

Las funciones también pueden ser llamados con argumentos clave de la forma clave = valor.

```
ask_ok(retries=5  ask="What do we do?")
```

Lambda

Por demanda popular, una serie de características que se encuentran comúnmente en los lenguajes de programación funcionales como Lisp se han añadido a Python. Con la palabra clave lambda, se pueden crear pequeñas funciones anónimas. He aquí una función que devuelve la suma de sus dos argumentos: lambda a, b: a + b.

La formas lambda pueden utilizarse siempre que se requieren los objetos de función. Están sintácticamente restringidas a una sola expresión. Semánticamente, no son más que otra manera de expresar una definición de función normal. Al igual que la definición de funciones anidadas, las formas lambda pueden hacer referencia a variables del ámbito contenedor.

```
>>> def make_incrementor(n):
... return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Listas arbitrarias de argumentos

Por último, la opción que se utiliza con menor frecuencia es especificar que una función puede ser llamada con un número arbitrario de argumentos. Estos argumentos serán envueltos en una tupla (ver Tuplas y secuencias). Antes del número variable de argumentos, puede aparecer cero o más argumentos normales.

```
def write_multiple_items(file, separator, \*args):
    file.write(separator.join(args))
```

2.8. Tuplas

Las tuplas, como las cadenas, son inmutables: no es posible asignar a los elementos individuales de una tupla (se puede simular la mayor parte de el mismo efecto con

corte y concatenación, sin embargo). También es posible crear tuplas que contienen objetos mutables, como las listas.

2.9. Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements.

```
>>> a = set('abracadabra')
>>> a #Mostrar por letra
set(['a', 'r', 'b', 'c', 'd'])
```

2.10. Diccionarios

Otro tipo de datos útiles incorporado en Python son los diccionarios. Los diccionarios se encuentran a veces en otros idiomas como "memorias asociativas" o "matrices asociativas". A diferencia de las secuencias, que están indexadas en un rango de números, los diccionarios se indexan por teclas, que puede ser cualquier tipo inmutable; cadenas y números siempre pueden ser teclas. Las tuplas pueden usarse como claves si sólo contienen cadenas, números o tuplas, si una tupla contiene cualquier objeto mutable directa o indirectamente, no puede ser utilizado como una clave.

Lo mejor es pensar en un diccionario como un conjunto desordenado de parejas clave: valor, con el requisito de que las claves son únicas (dentro de un diccionario).

Un par de llaves crea un diccionario vacío: {}. La colocación de una lista separada por comas de parejas *clave: valor* dentro de las llaves añade clave inicial: pares de valores al diccionario, lo que es también la forma en diccionarios se escriben en la salida.

He aquí un pequeño ejemplo de cómo usar un diccionario:

```
>>> tel = {'jack': 4098, 'sape': 4139} #Creamos el diccionario
>>> tel['guido'] = 4127                #Agregamos conjunto
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098} #El dicionario mas el nuevo conjunto
>>> tel['jack']
4098
>>> del tel['sape']                    #Eliminamos el conjunto
>>> tel['irv'] = 4127                  #Agregamos conjunto
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098} #El dicionario mas el nuevo conjunto y sin el campo eliminado
>>> tel.keys()             #Que muestre solo las clave de los conjuntos
['guido', 'irv', 'jack']
>>> 'guido' in tel          #Que diga si el conjunto guido pertenece al diccionario tel
True
```

Cuandoun un bucle recorre a través de los diccionarios, el valor de clave correspondiente se puede recuperar en el mismo método de tiempo utilizando los iteritems ().

```
>>> caballeros = {'jose': 'el inteligente', 'jesus': 'el audaz'} #Creas el diccionario
>>> for k, v in caballeros.iteritems():
... print k, v                                          #Indicas que imprima cada valor por separado
...
jose el inteligente
jesus el audaz
```

Cuando usas un bucle a través de una secuencia, el índice de posición y el valor correspondiente se pueden recuperar al mismo tiempo utilizando la función enumerate ().

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
... print i, v
...
0 tic
1 tac
2 toe
```

2.11. Módulos

Si sale del intérprete de Python y entrar de nuevo, las definiciones que ha hecho (funciones y variables) se pierden. Por lo tanto, si desea escribir un programa un poco más largo, es mejor usar un editor de texto para preparar la entrada del intérprete y ejecutarlo con ese archivo como insteadn entrada.

Un módulo es un archivo que contiene las definiciones y declaraciones de Python. El nombre del archivo es el nombre del módulo con el sufijo. Py adjunta. Por ejemplo, use su editor de texto favorito para crear un archivo llamado fibo.py en el directorio actual con el siguiente contenido:

```
def fib(n): # escribir la serie Fibonacci hasta n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
def fib2(n): # retorne la serie de Fibonacci hasta n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

A continuación, introduzca el intérprete de Python e importar este módulo con el siguiente comando:

```
>>> import fibo
```

2.12. Clases

Mecanismo de la clase de Python, añade clases al lenguaje con un mínimo de nueva sintaxis y la semántica.

Las características más importantes de las clases se mantienen con pleno poder, sin embargo: el mecanismo de herencia de clases permite múltiples clases, una clase derivada puede sobreescribir cualquier método de la clase base o de clases, y un método puede llamar al método de la clase base con el mismo nombre. Los objetos pueden contener una cantidad arbitraria de datos.

Sintaxis de definición de clase

La forma más simple de definición de una clase es asi:

```
class nombrecase:
```

Las definiciones de clases, como las definiciones de funciones (sentencias def) deben ser ejecutados antes de que tengan algún efecto. (Es posible que usted podría colocar una definición de clase en una rama de una sentencia if, o dentro de una función.)

Clases de objetos.

3. Interpretador Python

4.