Technical Training - Solutions

Release 7.0

OpenERP

2013-03-18

Contents

Based on a real case, this chapter covers: building an OpenERP module and its interface, Views, Reports, Workflows, Security aspects, Wizards, WebServices, Internationalization, Rapid Application Development (RAD) and Performance Optimization.

1 About OpenERP

1.1 Functional point of view

- Show OpenERP
- **Show main functional concepts on a demo db**
    - Menus
    - **Views**
        - List + Search
        - Form
        - Calendar, Graph, etc.
    - Modules
    - Users
    - Companies?
    - **Main objects (in a functional way!)**
        - Partner
        - Product

1.2 Technical point of view

- **General structure, schema**
    - Server - Client - Web Client
    - Communication: Net-RPC, XML-RPC
- **Framework - ORM**
    - What is an ORM? - Object, relations between objects
    - **OpenERP ORM particularities**
        - 1 instance of a logic object **is not** 1 record in the DB
        - 1 instance of a logic object is 1 table in the DB
        - Quick overview of OSV object, ORM methods

1.3 Installation on a UNIX Box for development purpose

Bazaar

- Versioning: general principle
- Centralised versioning system vs. Distributed versioning system
- Main BZR commands

OpenERP download and installation

OErp installation of trunk version on linux box:

- **Introduction to BZR first**
  - Bzr Installation
  - Main commands
  - Details on how they will use it this week
- **Download de Server, Client, Addons**
  - Remarks on extra-addons, community-addons
- **Dependencies installation**
  - PostgreSQL : Installation, add new user
  - Openerp configuration : how to create openerp-serverrc
  - Python dependencies

2 Python Introduction

Python is an indentation-sensitive interpreted imperative/object-oriented language, using some features of declarative/functional languages (these features are out of the scope of the current training). Don't hesitate to go check the python tutorial on the official Python documentation: http://docs.python.org/tutorial/index.html Almost the whole of the current introduction comes from there.

2.1 Python Interpreter

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: quit().

In interactive mode it prompts for the next command with the primary prompt, usually three greater-than signs (>>>); for continuation lines it prompts with the secondary prompt, by default three dots (...).

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this if statement:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

On BSD'ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line

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

(Assuming that the interpreter is on the user's PATH) at the beginning of the script and giving the file an executable mode.

2.2 Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses can be used for grouping. For example:

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

The equal sign ('=') is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

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

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

Variables must be "defined" (assigned a value) before they can be used, or an error will occur:

```
>>> # try to access an undefined variable
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

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

The conversion functions to floating point and integer (float(), int() and long())

2.3 Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes or double quotes:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
```

```
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Two string literals next to each other are automatically concatenated; the first line above could also have been written word = 'Help' 'A'; this only works with two literals, not with arbitrary string expressions:

```
>>> 'str' 'ing'             #  <-  This is ok
'string'
>>> 'str'.strip() + 'ing'   #  <-  This is ok
'string'
>>> 'str'.strip() 'ing'     #  <-  This is invalid
```

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one. Like in Icon, substrings can be specified with the slice notation: two indices separated by a colon.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Indices may be negative numbers, to start counting from the right. For example:

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
'Hel'
```

But note that -0 is really the same as 0, so it does not count from the right!

```
>>> word[-0]     # (since -0 equals 0)
'H'
```

String and Unicode objects have one unique built-in operation: the % operator (modulo). This is also known as the string formatting or interpolation operator. Given format % values (where format is a string or Unicode object), % conversion specifications in format are replaced with zero or more elements of values.

If format requires a single argument, values may be a single non-tuple object. Otherwise, values must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary). The possible conversions are:

'%d'	Signed integer decimal.
'%e'	Floating point exponential format (lowercase).
'%E'	Floating point exponential format (uppercase).
'%f'	Floating point decimal format.

'%F' Floating point decimal format.

'%c' Single character (accepts integer or single character string).

'%s' String (converts any Python object using str()).

Here is an example.

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

2.4 if Statements

Perhaps the most well-known statement type is the if statement. For example:

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
More
```

There can be zero or more elif parts, and the else part is optional. The keyword 'elif' is short for 'else if'

2.5 for Statements

The for statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists).

2.6 Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

Unlike strings, which are immutable, it is possible to change individual elements of a list:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

The built-in function len() also applies to lists:

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

list.append(x) Add an item to the end of the list; equivalent to a[len(a):] = [x].

list.count(x) Return the number of times x appears in the list.

list.sort() Sort the items of the list, in place.

list.reverse() Reverse the elements list, which are placed from the last one to the first one.

List Comprehensions

List comprehensions provide a concise way to create lists without resorting to use of map(), filter() and/or lambda. The resulting list definition tends often to be clearer than lists built using those constructs. Each list comprehension consists of an expression followed by a for clause, then zero or more for or if clauses. The result will be a list resulting from evaluating the expression in the context of the for and if clauses which follow it. If the expression would evaluate to a tuple, it must be parenthesized.

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
```

```
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]  # error - parens required for tuples
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(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]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

List comprehensions are much more flexible than map() and can be applied to complex expressions and nested functions:

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

2.7 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)     # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)     # call it
>>> f100                 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
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
```

Functions can also be called using keyword arguments of the form keyword = value.

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

Lambda Forms

By popular demand, a few features commonly found in functional programming languages like Lisp have been added to Python. With the lambda keyword, small anonymous functions can be created. Here's a function that returns the sum of its two arguments: lambda a, b: a+b. Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda forms can reference variables from the containing scope:

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

Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see Tuples and Sequences). Before the variable number of arguments, zero or more normal arguments may occur.

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

2.8 Tuples

Tuples, like strings, are immutable: it is not possible to assign to the individual items of a tuple (you can simulate much of the same effect with slicing and concatenation, though). It is also possible to create tuples which contain mutable objects, such as lists.

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                                  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
```

2.10 Dictionaries

Another useful data type built into Python is the dictionary (see Mapping Types — dict). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.

It is best to think of a dictionary as an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary).

A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the iteritems() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

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

2.11 Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead.

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

2.12 Classes

Python's class mechanism adds classes to the language with a minimum of new syntax and semantics.

The most important features of classes are retained with full power, however: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain an arbitrary amount of data.

Class Definition Syntax

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

Then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment.

The method function is declared with an explicit first argument representing the object, which is provided implicitly by the call.

Often, the first argument of a method is called self. This is nothing more than a convention: the name self has absolutely no special meaning to Python.

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new instance of the class and assigns this object to the local variable x.

The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if x is the instance of MyClass created above, the following piece of code will print the value 16, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Usually, a method is called right after it is bound:

```
x.f()
```

In the MyClass example, this will return the string ‘hello world’. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print xf()
```

will continue to print hello world until the end of time.

What exactly happens when a method is called? The special thing about methods is that the object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x). In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s object before the first argument.

Inheritance

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
    ...
```

3 Configuration

This document contains step-by-step solutions for the corresponding exercise booklet accompanying OpenERP Technical trainings.

Each section gives the solution to the exercises of the corresponding section in the exercise booklet. Step-by-step instructions are given in the simpliest possible form, usually giving an example of source code enabling to implement the required behaviour.

3.1 Open Source RAD with OpenObject

OpenERP is a modern Enterprise Management Software, released under the AGPL license, and featuring CRM, HR, Sales, Accounting, Manufacturing, Inventory, Project Management, ... It is based on OpenObject, a modular, scalable, and intuitive Rapid Application Development (RAD) framework written in Python.

- **OpenObject** features a complete and modular toolbox for quickly building applications: integrated Object-Relationship Mapping (ORM) support, template-based Model-View-Controller (MVC) interfaces, a report generation system, automated internationalization, and much more.

- **Python** is a high-level dynamic programming language, ideal for RAD,

combining power with clear syntax, and a core kept small by design.

Tip:

Useful links

- *Main website, with OpenERP downloads:* `www.openerp.com`
- *Functional & technical documentation:* `doc.openerp.com/trunk`
- *Community resources:* `www.launchpad.net/openobject`
- *Integration server:* `demo.openerp.com`
- *OpenERP E-Learning platform:* `edu.openerp.com`
- *Python doc:* `docs.python.org/2.6/`
- *PostgreSQL doc:* `www.postgresql.org/docs/8.4/static/index.html`

3.2 Installing OpenERP

OpenERP is distributed as packages/installers for most platforms, but can of course be installed from the source on any platform.

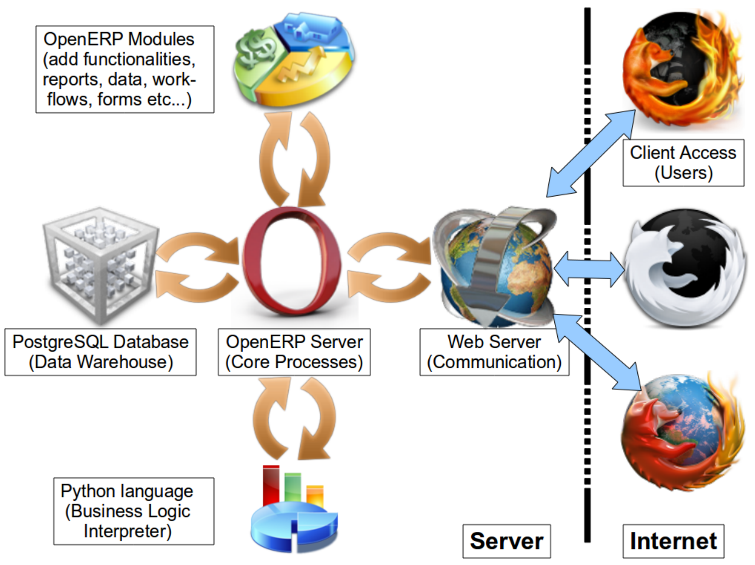
Note:

The procedure for installing OpenERP is likely to evolve (dependencies and so on), so make sure to always check the specific documentation (packaged & on website) for the latest procedures. See http://doc.openerp.com/install.

OpenERP Architecture

OpenERP uses the well-known client-server paradigm, with different pieces of software acting as client and server depending on the desired configuration.

OpenERP provides a web interface accessible using any modern browser.


OpenERP Modules (add functionalities, reports, data, work-flows, forms etc...)
Client Access (Users)
PostgreSQL Database (Data Warehouse)
OpenERP Server (Core Processes)
Web Server (Communication)
Python language (Business Logic Interpreter)
Server
Internet

3.3 Package installation

Windows	All-in-one installer, and separate installers for server, client, and webserver are on the website
Linux	openerp-server and openerp-client packages are available via corresponding package manager (e.g. Synaptic on Ubuntu) OR using BaZaar *bzr branch lp:openerp* (or *openerp/trunk* for the trunk version) when identified on Launchpad, then *cd openerp* (*cd trunk* in the trunk version) and *./bzr_set.py*

3.4 Installing from source

There are two alternatives:

1. using a tarball provided on the website, or
2. directly getting the source using Bazaar (distributed Source Version Control).

You also need to install the required dependencies (PostgreSQL and a few Python libraries - see documentation on `doc.openerp.com`).

Note:

OpenERP is Python-based, no compilation step is needed.

Typical bazaar checkout procedure (on Debian-based Linux)

```
$ sudo apt-get install bzr            # install bazaar version control
$ bzr branch lp:openerp-tools         # retrieve source installer
$ sh setup.sh                         # fetch code and perform setup
```

3.5 Database creation

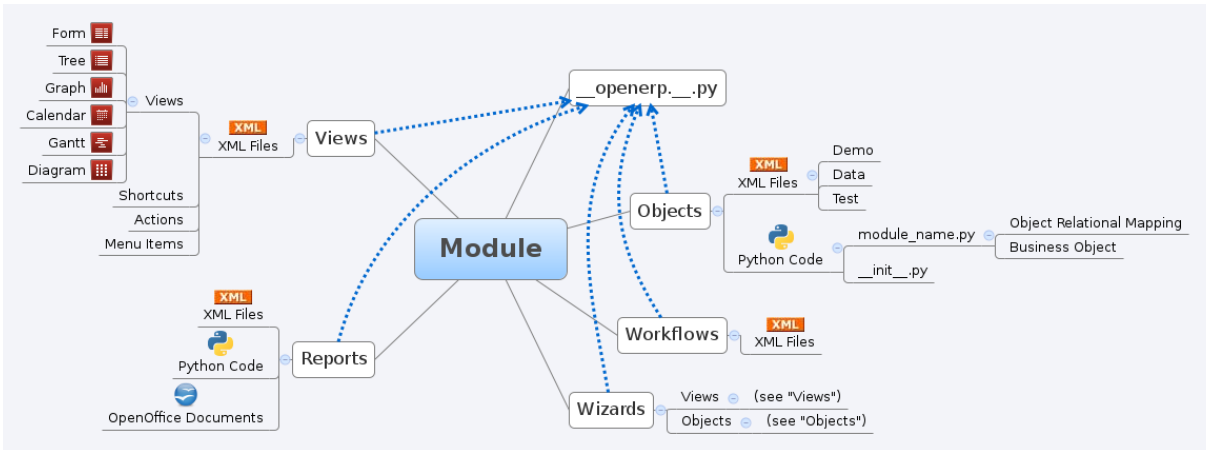
After installation, run the server and the client. From the login screen of the web client, click on *Manage Databases* to create a new database (default super admin password is admin). Each database has its own modules and configuration.

Note:

Demonstration data can also be included.

4 Build an OpenERP module

4.1 Composition of a module



A module can contain the following elements :

- **Business object :** declared as Python classes extending the OpenObject class osv.Model, the persistence of these resource is completly managed by OpenObject,
- **Data :** XML/CSV files with meta-data (views and workflows declaration), configuration data (modules parametrization) and demo data (optional bu recommended for testing),
- **Wizards :** stateful interactive forms used to assist users, often available as contextual actions on resources,
- **Reports :** RML (XML format). MAKO or OpenOffice report templates, to be merged with any kind of business data, and generate HTML, ODT or PDF reports.

4.2 Module Structure

Each module is contained in its own directory within either the `server/bin/addons` directory or another directory of addons, configured in server installation.

Note:

By default, the only directory of addons known by the server is `server/bin/addons`*. It is possible to add new addons by 1) copying them in* `server/bin/addons`*, or creating a symbolic link to each of them in this directory, or 2) specifying another directory containing addons to the server. The later can be accomplished either by running the server with the* –addons-path= *option, or by configuring this option in the* openerp_serverrc *file, automatically generated under Linux in your home directory by the server when executed with the* –save *option. You can provide several addons to the* addons_path = *option, separating them using commas.*

The *__init__.py* file is the Python module descriptor, because an OpenERP module is also a regular Python module. It contains the importation instruction applied to all Python files of the module, without the *.py* extension. For example, if a module contains a single python file named *mymodule.py*:

```
import mymodule
```

The *__openerp__.py* file is really the declaration of the OpenERP module. It is mandatory in each module. It contains a single python dictionary with several important pieces of informations, such as the name of the module, its description, the list of other OpenERP modules the installation of which is required for the current module to work properly. It also contains, among other things, a reference to all the data files (xml, csv, yml...) of the module. Its general structure is the following (see official documentation for a full file description):

```
{
    "name": "MyModule",
    "version": "1.0",
    "depends": ["base"],
    "author": "Author Name",
    "category": "Category",
    "description": """\
Description text
""",
    'data': [
        'mymodule_view.xml',
        #all other data files, except demo data and tests
    ],
    'demo': [
        #files containg demo data
    ],
    'test': [
        #files containg tests
    ],
    'installable': True,
    'auto_install': False,
}
```

Exercise 1 - Module Creation

Create the empty module Open Academy, with a *__openerp__.py* file. Install it under OpenERP.

1. Create a new folder “openacademy” somewhere on your computer.
2. Create an empty “__init__.py”
3. Create a “__openerp__.py” file under the “OpenAcademy” project:

```
{

    'name' : "Open Academy",
    'category' : "Test",
    'version' : "1.0",
    'depends' : ['base'],
    'author' : "Me",

    'description' : """\
Open Academy module for managing trainings:
    - training courses
    - training sessions
    - attendees registration""",

    'data' : [
    ],

}
```

4.3 Object Service - ORM

Key component of OpenObject, the Object Service (OSV) implements a complete Object-Relational mapping layer, freeing developers from having to write basic SQL plumbing. Business objects are declared as Python classes inheriting from the osv.Model class, which makes them part of the OpenObject Model, and magically persisted by the ORM layer.

Predefined osv.Model attributes for business objects

_name (required)	business object name, in dot-notation (in module namespace)
_columns (required)	dictionary {field names > object fields declarations }
_defaults	dictionary: { field names > functions providing defaults } _defaults['name'] = lambda self,cr,uid,context: 'eggs'
_rec_name	Alternative field to use as name, used by osv's name_get() (default: 'name')
...	...

4.4 ORM field types

Objects may contain three types of fields: simple, relational, and functional. Simple types are integers, floats, booleans, strings, etc. Relational fields represent the relationships between objects (one2many, many2one, many2many). Functional fields are not stored in the database but calculated on-the-fly as Python functions.

Common attributes supported by all fields (optional unless specified)

- **string** : Field label (required)
- **required** : True if mandatory
- **readonly** : True if not editable
- **help** : Help tooltip
- **select** : 1 to include in search views and optimize for list filtering (with database index) business object name, in dot-notation (in module namespace)
- **context** : Dictionary with contextual parameters (for relational fields)

Simple fields

- **boolean(...) integer(...) date(...) datetime(...) time(...)**
  - 'start_date': fields.date('Start Date')
  - 'active': fields.boolean('Active')
  - 'priority': fields.integer('Priority')
- **char(string,size,translate=False,..) text(string, translate=False,...)** [Text-based fields]
  - translate: True if field values can be translated by users
  - size: maximum size for char fields (→41,45)
- **float(string, digits=None, ...)** [Floating-point value with arbitrary precision and scale]
  - digits: tuple (precision, scale) (→58) . If digits is not provided, it's a float, not a decimal type.

4.5 Special/Reserved field names

A few field names are reserved for pre-defined behavior in OpenObject. Some of them are created automatically by the system, and in that case any field with that name will be ignored.

id	unique system identifier for the object (created by ORM, do not add it)
name	defines the value used by default to display the record in lists, etc. if missing, set _rec_name to specify another field to use for this purpose
...	See technical Memento for further details

Exercise 2 - Define a model

Define a new data model *Course* in the module openacademy. A course has a name, or “title”, and a description. The course name is required.

1. Under the new project “OpenAcademy”, create a new file “course.py”:

```
from openerp.osv import osv, fields

class Course(osv.Model):
    _name = "openacademy.course"

    _columns = {
        'name' : fields.char(string="Title", size=256, required=True),
        'description' : fields.text(string="Description"),
    }
```

2. Create an “__init__.py” file under the “OpenAcademy” project:

```
import course
```

4.6 Actions and Menus

Actions are declared as regular records and can be triggered in three ways:

1. by clicking on menu items linked to a specific action
2. by clicking on buttons in views, if these are connected to actions
3. as contextual actions on an object

Menus are also regular records in the ORM. However, they are declared with the tag `<menuitem>`. This is a shortcut for declaring an “ir.ui.menu” record, and connect it with a corresponding action via an “ir.model.data” record.

The following example defines a menu item to display the list of ideas. The action associated to the menu mentions the model of the records to display, and which views are enabled; in this example, only the tree and form views will be available. There are other optional fields for actions, see the documentation for a complete description of them.

```
<record model="ir.actions.act_window" id="action_list_ideas">
    <field name="name">Ideas</field>
    <field name="res_model">idea.idea</field>
    <field name="view_mode">tree,form</field>
    <field name="help" type="html">
        <p class="oe_view_nocontent_create">A nice arrow with some help for
        your first record</p>
    </field>
</record>

<menuitem id="menu_ideas" parent="menu_root" name="Ideas" sequence="10"
          action="action_list_ideas"/>
```

Note:

The action is usually declared before its corresponding menu in the XML file.
This is because the record “action_id” must exist in the database to allow the ORM to create the menu record.

Exercise 3 - Define new menu entries

Define new menu entries to access courses and sessions under the OpenAcademy menu entry; one should be able to 1) display a list of all the courses and 2) create/modify new courses.

1. Create view/openacademy.xml:

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

  <!-- This starts a new comment.
       ... and this one, end it -->

  <!-- All data in OpenERP are stored into /openerp/data branch
       ... which means in XML:

       <openerp>
           <data>
           </data>
       </openerp>

       -->

  <!-- Indenting in XML is optional. but
       please try to stay human readable -->

  <data>

    <!-- action windows -->
    <!-- The following tag is an action definition.
         Basically, we create a record in model ir.actions.act_window
         OpenERP will do the rest -->
    <record model="ir.actions.act_window" id="course_list_action">
      <field name="name">Courses</field>
      <field name="res_model">openacademy.course</field>
      <field name="view_type">form</field>
      <field name="view_mode">tree,form</field>
      <field name="help" type="html">
        <p class="oe_view_nocontent_create">Create the first course</p>
      </field>
    </record>

    <!-- menuitems -->
    <menuitem id="main_openacademy_menu" name="Open Academy" />

    <!-- A first level in the left side menu is needed
         before using action= attribute -->
    <menuitem id="openacademy_menu" name="Open Academy"
              parent="main_openacademy_menu" />

    <!-- the following menuitem should appear *after*
         its parent openacademy_menu and *after* its
         action course_list_action -->
    <menuitem id="courses_menu" name="Courses"
              parent="openacademy_menu"
              action="course_list_action" />

    <!-- Full id location:
         action="openacademy.course_list_action"
         It is not required when it is the same module -->
  </data>
</openerp>
```

2. In “__openerp__.py”:

```
{

    'name' : "Open Academy",

    (...)

    'data' : [
        'view/openacademy.xml',
    ],

}
```

5 Building views: basics

Views form a hierarchy. Several views of the same type can be declared on the same object, and will be used depending on their priorities.

It is also possible to add/remove elements in a view by declaring an inherited view (see *View inheritance*).

5.1 Generic view declaration

A view is declared as a record of the model *ir.ui.view*. Such a record is declared in XML as

```
<record model="ir.ui.view" id="view_id">
    <field name="name">view.name</field>
    <field name="model">object_name</field>
    <field name="priority" eval="16"/>
    <field name="arch" type="xml">
        <!-- view content: <form>, <tree>, <graph>, ... -->
    </field>
</record>
```

Notice that the content of the view is itself defined as XML. Hence, the field *arch* of the record must be declared of type XML, in order to parse the content of that element as the field’s value.

5.2 Tree views

Tree views, also called list views, display records in a tabular form. They are defined with the XML element `<tree>`. In its simplest form, it mentions the fields that must be used as the columns of the table.

```
<tree string="Idea list">
    <field name="name"/>
    <field name="inventor_id"/>
</tree>
```

5.3 Form views

Forms allow the creation/edition of resources, and are defined by XML elements `<form>`. The following example shows how a form view can be spatially structured with separators, groups, notebooks, etc. Consult the documentation to find out all the possible elements you can place on a form.

```xml
<form string="Idea form">
    <group colspan="2" col="2">
        <separator string="General stuff" colspan="2"/>
        <field name="name"/>
        <field name="inventor_id"/>
    </group>
    <group colspan="2" col="2">
        <separator string="Dates" colspan="2"/>
        <field name="active"/>
        <field name="invent_date" readonly="1"/>
    </group>

    <notebook colspan="4">
        <page string="Description">
            <field name="description" nolabel="1"/>
        </page>
    </notebook>

    <field name="state"/>
</form>
```

Now with the new version 7 you can write html in your form

```xml
<form string="Idea Form v7"  version="7.0">
    <header>
        <button string="Confirm" type="object"
                name="action_confirm"
                states="draft"  class="oe_highlight" />
        <button string="Mark as done" type="object"
                name="action_done"
                states="confirmed" class="oe_highlight"/>
        <button string="Reset to draft" type="object"
                name="action_draft"
                states="confirmed,done" />
        <field name="state" widget="statusbar"/>
    </header>
    <sheet>
        <div class="oe_title">
            <label for="name" class="oe_edit_only" string="Idea Name" />
            <h1><field name="name" /></h1>
        </div>
        <separator string="General" colspan="2" />
        <group colspan="2" col="2">
            <field name="description" placeholder="Idea description..." />
        </group>
    </sheet>
</form>
```

Exercise 1 - Customise form view using XML

Create your own form view for the *Course* object. Data displayed should be: the name and the description of the course.

1. In openacademy_view.xml create the following views:

```xml
<!-- This is the form declaration -->
<record model="ir.ui.view" id="course_form_view">
  <field name="name">course.form</field>
  <field name="model">openacademy.course</field>
  <field name="arch" type="xml">
    <form string="Course Form">
      <field name="name" />
      <field name="description" />
    </form>
  </field>
</record>
```

Note:

By default, the view is separated in 4 columns

2. We can improve the form view for a better display.

```
<!-- This is the form declaration -->
<record model="ir.ui.view" id="course_form_view">
  <field name="name">course.form</field>
  <field name="model">openacademy.course</field>
  <field name="arch" type="xml">
    <form string="Course Form">
      <!-- Note: default form columns are 4
                 ... so, defining a colspan of 4 will
                     stretch the field in the entire form -->
      <field name="name" colspan="4" />
      <field name="description" colspan="4" />
    </form>
  </field>
</record>
```

Exercise 2 - Notebooks

In the Course form view, put the description field under a tab, such that it will be easier to add other tabs later, containing additional information.

Modify the Course form view as follows:

```
<!-- This is the form declaration -->
<record model="ir.ui.view" id="course_form_view">
  <field name="name">course.form</field>
  <field name="model">openacademy.course</field>
  <field name="arch" type="xml">
    <form string="Course Form">
      <!-- Note: default form columns are 4
                 ... so, defining a colspan of 4 will
                     stretch the field in the entire form -->
      <field name="name" colspan="4" />
      <!-- "notebook"... uh? Forget that, it's just tabs -->
      <notebook colspan="4">
        <page string="Description">
          <field name="description" colspan="4" nolabel="1" />
        </page>
        <page string="About">
          <label string="This is an example of notebooks" />
        </page>
      </notebook>
    </form>
  </field>
</record>
```

6 Relations between Objects

Exercise 1 - Create classes

Create the classes *Session* and *Attendee*, and add a menu item and an action to display the sessions. A session has a name (required), a start date, a duration (in days) and a number of seats. An attendee has a name, which is not required.

1. Create classes *Session* and *Attendee*.

```
class Session(osv.Model):
    _name = "openacademy.session"

    _columns = {
        'name' : fields.char(string="Name", size=256, required=True),
        'start_date' : fields.date(string="Start date"),
        'duration' : fields.float(string="Duration", digits=(6,2),
                                  help="Duration in days"),
        'seats' : fields.integer(string="Number of seats"),
    }

class Attendee(osv.Model):
    _name = "openacademy.attendee"

    _columns = {
        'name' : fields.char(string="Name", size=256),
    }
```

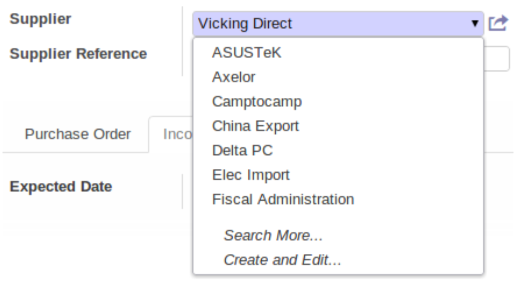
Note:

digits=(6,2) *specifies the precision of a float number: 6 is the total number of digits, while 2 is the number of digits after the comma. Note that it results that the number of digits before the comma is maximum 4 in this case.*

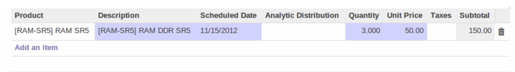
6.1 Relational fields

Relational fields are fields with values that refer to other objects.

- Common attributes supported by relational fields
  - domain: optional restriction in the form of arguments for search (see search())
- **many2one(obj, ondelete='set null',...)** : simple relationship towards another object (using a foreign key)



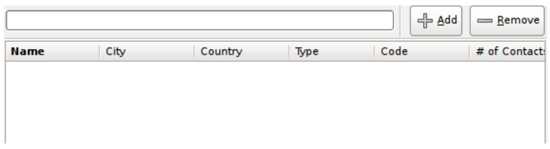
  - obj: _name of destination object (required)
  - ondelete: deletion handling: 'set null', 'cascade', 'restrict' (see PostgreSQL documentation).
- **one2many(obj, field_id, ...)** : virtual relationship towards multiple objects (inverse of many2one). A one2many relational field provides a kind of "container" relation: the records on the other side of the relation can be seen as "contained" in the record on this side.



Product	Description	Scheduled Date	Analytic Distribution	Quantity	Unit Price	Taxes	Subtotal
[RAM-SR5] RAM SR5	[RAM-SR5] RAM DDR SR5	11/15/2012		3.000	50.00		150.00
Add an item							

  - obj: _name of destination object (required)
  - **field_id: field name of inverse many2one, i.e., corresponding foreign** key (required)

- **many2many(obj, rel, field1, field2, ...)** : bidirectional multiple relationship between objects. This is the most general kind of relation: a record may be related to any number of records on the other side, and vice-versa.

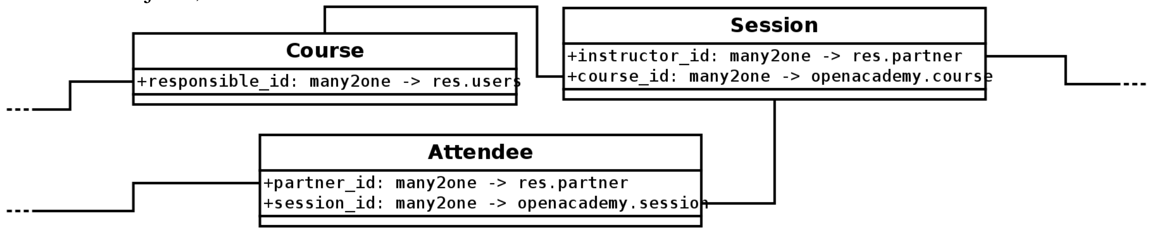


  - obj: _name of destination object (required)
  - rel: relationship table to use
  - field1: name of field in rel table storing the id of the current object
  - field2: name of field in rel table storing the id of the target object

  Note that the parameters rel, field1 and field2 are optional. When they are not given, the ORM automatically generates suitable values for them.

Exercise 2 - Relations many2one

Using a many2one relational field, modify the classes Course, Session and Attendee to reflect their relations with other objects, defined as follows.


Course
+responsible_id: many2one -> res.users
Session
+instructor_id: many2one -> res.partner
+course_id: many2one -> openacademy.course
Attendee
+partner_id: many2one -> res.partner
+session_id: many2one -> openacademy.session

1. Modify the classes as follows.

```
class Course(osv.Model):
    _name = "openacademy.course"

    _columns = {
        'name' : fields.char(string="Title", size=256, required=True),
        'description' : fields.text(string="Description"),

        # Relational fields
        'responsible_id' : fields.many2one('res.users',
            # 'set null' will reset responsible_id to
            # undefined if responsible was deleted
            ondelete='set null', string='Responsible', select=True),
    }

class Session(osv.Model):
    _name = "openacademy.session"

    _columns = {
        'name' : fields.char(string="Name", size=256, required=True),
        'start_date' : fields.date(string="Start date"),
        'duration' : fields.float(string="Duration", digits=(6,2),
                                  help="Duration in days"),
        'seats' : fields.integer(string="Number of seats"),

        # Relational fields
        'instructor_id' : fields.many2one('res.partner', string="Instructor"),
        'course_id' : fields.many2one('openacademy.course',
            # 'cascade' will destroy session if course_id was deleted
            ondelete='cascade', string="Course", required=True),
    }

class Attendee(osv.Model):
    _name = "openacademy.attendee"

    # _rec_name redefine the field to get when seeing the record in
    # another object. In this case, the partner's name will be printed
    _rec_name = 'partner_id'
    # (useful when no field 'name' has been defined)

    _columns = {
        # There is only relational fields
        'partner_id': fields.many2one('res.partner', string="Partner",
                                      required=True, ondelete='cascade'),

        'session_id': fields.many2one('openacademy.session', string="Session",
                                      required=True, ondelete='cascade'),
    }
```

2. add access to the session object in `view/openacademy.xml`

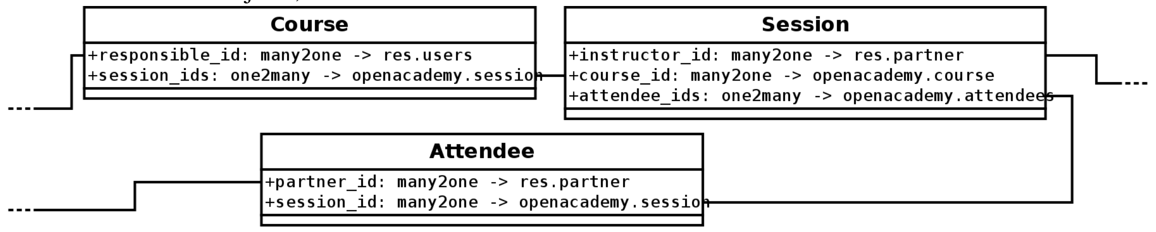
```
<record model="ir.actions.act_window" id="session_list_action">
  <field name="name">Sessions</field>
  <field name="res_model">openacademy.session</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form</field>
</record>

(...)

<menuitem id="session_menu" name="Sessions"
          parent="openacademy_menu"
          action="session_list_action" />
```

Note:

In the class Attendee, we have removed the field name. *Instead of that field, the name of the partner will be used to identify the Attendee record. This is what the attribute* _rec_name *is used for.*

Exercise 3 - Inverse relation one2many

Using the inverse relational field one2many, modify the classes Course, Session and Attendee to reflect their relations with other objects, defined as follows.


Course
+responsible_id: many2one -> res.users
+session_ids: one2many -> openacademy.session
Session
+instructor_id: many2one -> res.partner
+course_id: many2one -> openacademy.course
+attendee_ids: one2many -> openacademy.attendees
Attendee
+partner_id: many2one -> res.partner
+session_id: many2one -> openacademy.session

1. Modify the classes as follows.

```
class Course(osv.Model):
    _name = "openacademy.course"

    _columns = {
        'name' : fields.char(string="Title", size=256, required=True),
        'description' : fields.text(string="Description"),

        # Relational fields
        'responsible_id' : fields.many2one('res.users',
            # 'set null' will reset responsible_id to
            # undefined if responsible was deleted
            ondelete='set null', string='Responsible', select=True),
        # A one2many is the inverse link of a many2one
        'session_ids' : fields.one2many('openacademy.session', 'course_id',
                                        string='Session'),
    }

class Session(osv.Model):
    _name = "openacademy.session"

    _columns = {
        'name' : fields.char(string="Name", size=256, required=True),
        'start_date' : fields.date(string="Start date"),
        'duration' : fields.float(string="Duration", digits=(6,2),
                                  help="Duration in days"),
        'seats' : fields.integer(string="Number of seats"),

        # Relational fields
        'instructor_id' : fields.many2one('res.partner', string="Instructor"),
        'course_id' : fields.many2one('openacademy.course',
            # 'cascade' will destroy session if course_id was deleted
            ondelete='cascade', string="Course", required=True),
        'attendee_ids' : fields.one2many('openacademy.attendee', 'session_id',
                                         string="Attendees"),
    }
```

Note:

The definition of a one2many *field* ***requires*** *the corresponding* many2one *field to be defined in the referred class. This* many2one *field name is given as second argument to the* one2many *field (here,* course_id*).*

Exercise 4 - Modify the views

Modify/Create the tree and form views for the *Course* and *Session* objects. Data displayed should be:
For the *Course* object:

- in the tree view, the name of the course and the responsible for that course;
- in the form view, the name and the responsible on the top, as well as the description of the course in one tab and, and the related sessions in a second tab.

For the *Session* object:

- in the tree view, the name of the session and the related course;
- in the form view, all the fields of the Session object.

Since the amount of data to display is quite large, try to arrange the form views in such a way that information looks clear.

1. Modify the Course views:

```
<!-- views -->
<!-- This is the form declaration -->
<record model="ir.ui.view" id="course_form_view">
  <field name="name">course.form</field>
  <field name="model">openacademy.course</field>
  <field name="arch" type="xml">
    <form string="Course Form">
      <!-- Note: default form columns are 4
                 ... so, defining a colspan of 4 will
                     stretch the field in the entire form -->
      <field name="name" colspan="4" />
      <field name="responsible_id" />
      <!-- "notebook"... uh? Forget that, it's just tabs -->
      <notebook colspan="4">
        <page string="Description">
          <field name="description" colspan="4" nolabel="1" />
        </page>
        <page string="About">
          <label string="This is an example of notebooks" />
        </page>
        <page string="Sessions">
          <!-- Let's make a form and tree view inside another form!
               We just need the <tree> and <form> tags to define what
               field we need -->
          <field name="session_ids" nolabel="1" colspan="4"
                 mode="tree">
            <tree string="Registered sessions">
              <field name="name"/>
              <field name="instructor_id"/>
            </tree>
          </field>
        </page>
      </notebook>
    </form>
  </field>
</record>

<!-- The declaration below will overwrite the default tree (or list) view
     of our openacademy.course model -->
<record model="ir.ui.view" id="course_tree_view">
  <field name="name">course.tree</field>
  <field name="model">openacademy.course</field>
  <field name="arch" type="xml">
    <tree string="Course Tree">
      <field name="name" />
      <!-- Remember that any tag who have no children can be written
           like above. But you can still write the full XML form
           like this if you want

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

           -->
      <field name="responsible_id" />
    </tree>
  </field>
</record>
```

Note:

In the above example, if the form view of the field session_ids *was not customized in order to show only the name and instructor of the session, the field* course_id *(as well as all the other fields of the session object) would appear, and would be required. However, whatever value would be entered in the field* course_id, *it would be ignored when the session instance is saved, and replaced by the* course_id *of the current instance of the course object.*

2. Create the Session views:

Note:

By default, the view is separated in 4 columns. We can improve the session form view for a better display by using the attribute colspan.

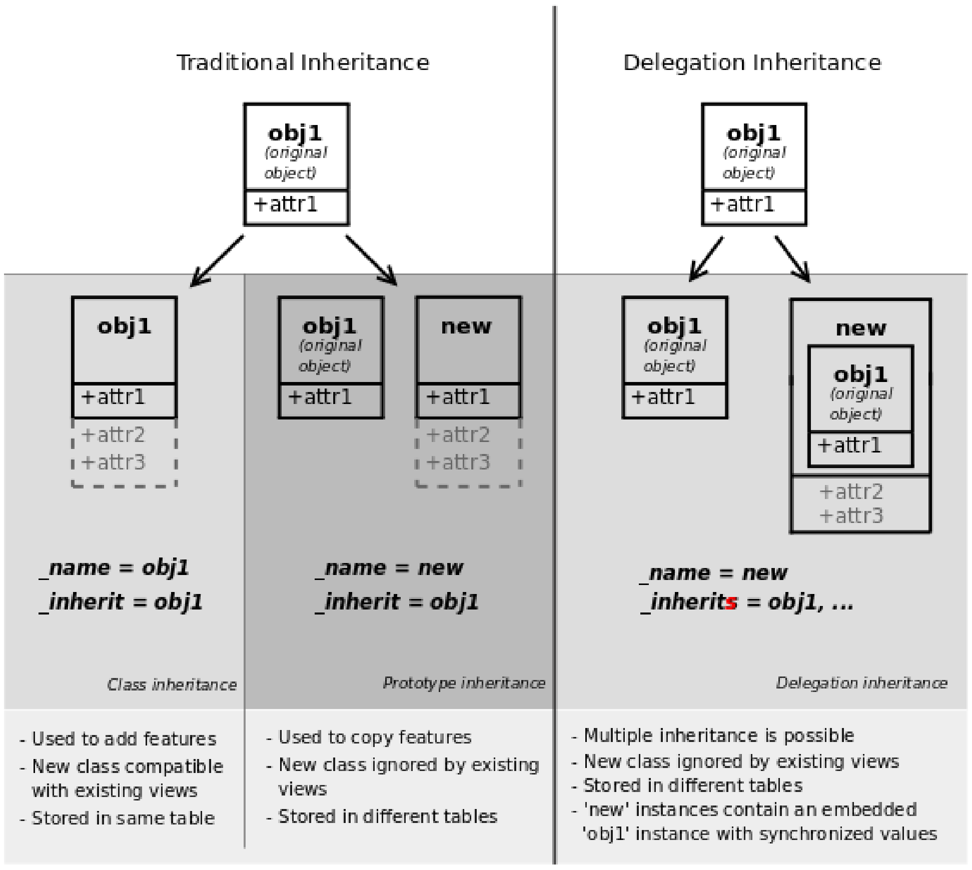
```
<!-- session's form view -->
<record model="ir.ui.view" id="session_form_view">
  <field name="name">session.form</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <form string="Session Form">
      <group colspan="2" col="2">
        <separator string="General" colspan="2"/>
        <field name="course_id"/>
        <field name="name" />
        <field name="instructor_id" />
      </group>
      <group colspan="2" col="2">
        <separator string="Schedule" colspan="2"/>
        <field name="start_date"/>
        <field name="duration"/>
        <field name="seats"/>
      </group>
      <separator string="Attendees" colspan="4"/>
      <field name="attendee_ids" colspan="4" nolabel="1">
        <!-- 'editable' attribute will set the position
             to push new elements in the list -->
        <tree string="" editable="bottom">
          <field name="partner_id"/>
        </tree>
      </field>
    </form>
  </field>
</record>

<!-- session's tree/list view -->
<record model="ir.ui.view" id="session_tree_view">
  <field name="name">session.tree</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <tree string="Session Tree">
      <field name="name"/>
      <field name="course_id"/>
    </tree>
  </field>
</record>
```

Note:

Here the groups appear as two columns; we could show them on two lines by setting the "colspan" of the groups to "4" or using a tag <newline/>.

7 Inheritance

7.1 Inheritance mechanisms


Traditional Inheritance
Delegation Inheritance
obj1 (original object) +attr1
obj1 +attr1 +attr2 +attr3
obj1 (original object) +attr1
new +attr1 +attr2 +attr3
new obj1 (original object) +attr1 +attr2 +attr3
_name = obj1
_inherit = obj1
_name = new
_inherit = obj1
_name = new
_inherits = obj1, ...
Class inheritance
Prototype inheritance
Delegation inheritance
- Used to add features
- New class compatible with existing views
- Stored in same table
- Used to copy features
- New class ignored by existing views
- Stored in different tables
- Multiple inheritance is possible
- New class ignored by existing views
- Stored in different tables
- 'new' instances contain an embedded 'obj1' instance with synchronized values

Predefined osv.Model attributes for business objects

_inherit	_name of the parent business object (for prototype inheritance)
_inherits	for multiple / instance inheritance mechanism: dictionary mapping the _name of the parent business objects to the names of the corresponding foreign key fields to use

7.2 View inheritance

Existing views should be modified through inherited views, and never directly. An inherited view references its parent view using the field inherit_id, and may add or modify existing elements in the view by referencing them through element selectors or XPath expressions (specifying the appropriate position).

position	• inside: placed inside match (default) • before: placed before match • attributes : to change attribute of the tag	• replace: replace match • after: placed after match

Tip:

XPath reference can be found at `www.w3.org/TR/xpath`

```
<!-- improved idea categories list -->
<record id="idea_category_list2" model="ir.ui.view">
    <field name="name">id.category.list2</field>
    <field name="model">ir.ui.view</field>
    <field name="inherit_id" ref="id_category_list"/>
    <field name="arch" type="xml">
        <!-- find field name, and insert field reference before it -->
        <field name="name" position="before">
            <field name="reference />
        </field>

        <!-- find field description inside tree, and add the field
             idea_ids after it -->
        <xpath expr="/tree/field[@name='description']" position="after">
            <field name="idea_ids" string="Number of ideas"/>
        </xpath>
    </field>
</record>
```

Exercise 1 - Add an inheritance mechanism

Using class inheritance, create a "Partner" class which modifies the existing "Partner" class, adding a "is_instructor" boolean field, and the list of the sessions this partner will attend to. Using views inheritance, modify the existing partners form view to display the new fields.

1. Create a "partner.py" file under the "OpenAcademy" project. Create the following class:

```python
from openerp.osv import osv, fields

class Partner(osv.Model):
    """Inherited res.partner"""

    # The line above is the Python's way to document
    # your objects (like classes)
    _inherit = 'res.partner'

    _columns = {
        # We just add a new column in res.partner model
        'instructor' : fields.boolean("Instructor"),
    }

    _defaults = {
        # By default, no partner is an instructor
        'instructor' : False,
    }
```

2. Create a "partner_view.xml" file under the "OpenAcademy" project. Create the following inherited views:

Note:

This part is the opportunity to introduce the developer mode to inspect the view to modify, find the xml_id of the view and the place to put the new field. You can activate it by putting "?debug" before the sharp "#".

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

    <!-- Add instructor field to existing view -->
    <record model="ir.ui.view" id="partner_instructor_form_view">
      <field name="name">partner.instructor</field>
```

```
        <field name="model">res.partner</field>
        <field name="inherit_id" ref="base.view_partner_form" />
        <field name="arch" type="xml">
          <field name="is_company" position="before">
            <field name="instructor" />
            <label for="instructor" string="Is an Instructor?" />
          </field>
      </record>

      <record model="ir.actions.act_window" id="contact_list_action">
        <field name="name">Contacts</field>
        <field name="res_model">res.partner</field>
        <field name="view_type">form</field>
        <field name="view_mode">tree,form</field>
      </record>

      <menuitem id="configuration_menu" name="Configuration"
                parent="main_openacademy_menu" />

      <menuitem id="contact_menu" name="Contacts"
                parent="configuration_menu"
                action="contact_list_action" />

    </data>
  </openerp>
```

Note:

The elements placed inside a parent field having an attribute position="after" *will be displayed in the view after the field identified by the value of the* name *attribute of the parent field. If there are more than one element, they will be displayed in* ***reverse order*** *of their appearance in the XML code.*

3. In "__init__.py" add the following line:

```
import partner
```

4. Add "view/partner.xml" to __openerp__.py's "data" key:

```
{
    (...)

    'data' : [
        # The order matter
        'view/openacademy.xml',
        'view/partner.xml',
    ],
}
```

Domains

Domains are used to select a subset of records from a model, according to a list of criteria. Each criteria is a tuple of three element: a field name, an operator (see the technical memento for details), and a value. For instance, the following domain selects all products of type service that have a unit_price greater than 1000.

```
[('product_type', '=', 'service'), ('unit_price', '>', 1000)]
```

By default, criteria are combined with the logic operator "AND".One can insert the logic operators `'&'`, `'|'`, `'!'` (respectively AND, OR, NOT) inside domains. They are introduced using a *prefix* notation, i.e., the operator is put before its arguments. The example below encodes the condition: "being of type service OR NOT having a unit price between 1000 and 2000".

```
['|', ('product_type', '=', 'service'),
    '!', '&', ('unit_price', '>=', '1000'), ('unit_price', '<', '2000')]
```

Exercise 2 - *domain*

Add a mechanism allowing the *Session* form view to allow the user to choose the instructor only among the partners the “Instructor” field of which is set to “True”.

You can either:

1. Modify the *Session* class as follows:

```
class Session(osv.Model):
    _name = "openacademy.session"

    _columns = {
        (...)
        # Relational fields
        'instructor_id' : fields.many2one('res.partner', string="Instructor",
            domain=[('instructor','=',True)]),
        (...)
    }
```

Note:

If the domain is declared as a list (instead of a string), it is evaluated server-side and cannot refer to dynamic values (like another field of the record). When the domain is declared as a string, it is evaluated client-side and dynamic values are allowed.

2. Or modify the *Session*‘s “instructor_id” field as follows:

```
<field name="instructor_id" domain="[('instructor','=',True)]"/>
```

Exercise 3 - *domain*

We now decide to create new categories among the partners: *Teacher/Teacher Level 1* and *Teacher/Teacher Level 2*. Modify the domain defined in the previous exercise to allow the user to choose the instructor among the partners the “Instructor” field of which is set to “True” **or** those who are in one of the categories we defined.

1. Modify the *Session* class as follows:

```
class Session(osv.Model):
    _name = "openacademy.session"

    _columns = {
        (...)

        # Relational fields
        'instructor_id' : fields.many2one('res.partner', string="Instructor",
            domain=['|',('instructor','=',True),
                        ('category_id.name','ilike','Teacher')]),
        (...)
    }
```

2. Modify “view/partner.xml” to get access to *Partner categories*:

```
(...)

<!-- Actions window -->
```

```
<record model="ir.actions.act_window" id="contact_cat_list_action">
  <field name="name">Contact tags</field>
  <field name="res_model">res.partner.category</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form</field>
</record>

<!-- Menus -->
(...)

<menuitem id="contact_cat_menu" name="Contact tags"
          parent="configuration_menu"
          action="contact_cat_list_action" />

(...)
```

8 ORM Methods

8.1 Functional fields

function(fnct, arg=None, fnct_inv=None, fnct_inv_arg=None, type='float', fnct_search=None, obj=None, store=False, multi=False, ...): Functional field simulating a real field, computed rather than stored

- **fnct** [function to compute the field value (required)]
    - def fnct(self, cr, uid, ids, field_name, arg, context) returns a dictionary { ids→values } with values of type type
- **fnct_inv** [function used to write a value in the field instead]
    - def fnct_inv(obj, cr, uid, id, name, value, fnct_inv_arg, context)
- **type** : type of simulated field (any other type besides 'function')
- **obj** : model _name of simulated field if it is a relational field
- **fnct_search** [function used to search on this field]
    - def fnct_search(obj, cr, uid, obj, name, args) returns a list of tuples arguments for search(), e.g. [('id','in',[1,3,5])]
- **store, multi** : optimization mechanisms (see usage in Performance Section)

related(f1, f2, ..., type='float', ...): Shortcut field equivalent to browsing chained fields

- **f1,f2,...** : chained fields to reach target (f1 required)
- **type** : type of target field
- **obj** : model _name of target field if it is a relational field

property(obj, type='float', view_load=None, group_name=None, ...): Dynamic attribute with specific access rights

- **obj** : object (required)
- **type** : type of equivalent field

Exercise 1 - Functional fields

Add a functional field to the "Session" class, that contains the percentage of taken seats in a session. Display that percentage in the session tree and form views. Once it is done, try to display it under the form of a progressbar.

1. In “course.py”, modify the Session class as follows:

```
class Session(osv.Model):
    _name = "openacademy.session"

    def _get_taken_seats_percent(self, seats, attendee_list):
        # Prevent divison-by-zero error
        try:
            # You need at least one float number to get the real value
            return (100.0 * len(attendee_list)) / seats
        except ZeroDivisionError:
            return 0.0

    # The following definition must appear *after* _get_taken_seats_percent
    # because it's called during the process
    def _taken_seats_percent(self, cr, uid, ids, field, arg, context=None):
        # Compute the percentage of seats that are booked
        result = {}
        for session in self.browse(cr, uid, ids, context=context):
            result[session.id] = self._get_taken_seats_percent(
                session.seats, session.attendee_ids)
        return result

    _columns = {
        'name' : fields.char(string="Name", size=256, required=True),
        (...)

        # Function fields are calculated on-the-fly when reading records
        'taken_seats_percent' : fields.function(_taken_seats_percent,
            type='float', string='Taken Seats'),

        # Relational fields
        'instructor_id' : fields.many2one('res.partner', string="Instructor",
        (...)
    }
```

2. In “course/openacademy.xml”, modify the *Session* view as follows:

```
<!-- session's form view -->
<record model="ir.ui.view" id="session_form_view">
  <field name="name">session.form</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <form string="Session Form">
      (...)
      <group colspan="2" col="2">
        <separator string="Schedule" colspan="2"/>
        <field name="start_date"/>
        <field name="duration"/>
        <field name="seats"/>
        <field name="taken_seats_percent" widget="progressbar"/>
      </group>
      (...)
    </form>
  </field>
</record>

<!-- session's tree/list view -->
<record model="ir.ui.view" id="session_tree_view">
  <field name="name">session.tree</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <tree string="Session Tree">
```

```
        (...)
        <field name="taken_seats_percent" widget="progressbar"/>
      </tree>
    </field>
</record>
```

8.2 Onchange

```
<!-- on_change example in xml -->
<field name="amount" on_change="onchange_price(unit_price,amount)" />
<field name="unit_price" on_change="onchange_price(unit_price,amount)" />
<field name="price" />

def onchange_price(self, cr, uid, ids, unit_price, amount, context=None):
    """ change the price when the unit price or the amount is changed """
    return { 'value' : {'price' : unit_price * amount}}
```

Exercise 2 - *Onchange* methods

Modify the Session form view and the Session class in such a way that the percentage of taken seats refreshes whenever the number of available seats or the number of attendees changes, without having to save the modifications.

1. Modify the *attendee_ids* and *seats* fields of the Session form view by adding a *on_change* attribute, as follows:

```
<field name="seats" on_change="onchange_taken_seats(seats,attendee_ids)">

<field name="attendee_ids" colspan="4" nolabel="1"
       on_change="onchange_taken_seats(seats,attendee_ids)">
```

2. Modify the Session class by adding the the following method:

```
class Session(osv.Model):
    _name = "openacademy.session"

    def _get_taken_seats_percent(self, seats, attendee_list):
        # Prevent divison-by-zero error
        try:
            # You need at least one float number to get the real value
            return (100.0 * len(attendee_list)) / seats
        except ZeroDivisionError:
            return 0.0

    # The following definition must appear \*after\* _get_taken_seats_percent
    # because it's called during the process
    def _taken_seats_percent(self, cr, uid, ids, field, arg, context=None):
        (...)

    # Same as the previous definition: it needs to be declared
    # *after* _get_taken_seats_percent
    def onchange_taken_seats(self, cr, uid, ids, seats, attendee_ids):
        # Serializes o2m commands into record dictionaries (as if all the o2m
        # records came from the database via a read()), and returns an
        # iterable over these dictionaries.
        attendee_records = self.resolve_2many_commands(
            cr, uid, 'attendee_ids', attendee_ids, ['id'])
        res = {
            'value' : {
                'taken_seats_percent' :
                    self._get_taken_seats_percent(seats, attendee_records),
            },
        }
        return res

    (...)
```

Note:

resolve_2many_commands *serializes o2m and m2m commands into record dictionaries (as if all the o2m records came from the database via a read()), and returns an iterable over these dictionaries.*

Exercise 3 - ***warning***

Modify this *onchange* function to raise a warning when the number of seats is under zero.

```
# Same as the previous definition: it needs to be declared
# *after* _get_taken_seats_percent
def onchange_taken_seats(self, cr, uid, ids, seats, attendee_ids):
    # Serializes o2m commands into record dictionaries (as if all the o2m
    # records came from the database via a read()), and returns an
    # iterable over these dictionaries.
    attendee_records = self.resolve_o2m_commands_to_record_dicts(
        cr, uid, 'attendee_ids', attendee_ids, ['id'])
    res = {
        'value' : {
            'taken_seats_percent' :
                self._get_taken_seats_percent(seats, attendee_records),
        },
    }
    if seats < 0:
        res['warning'] = {
            'title'   : "Warning: bad value",
            'message' : "You cannot have negative number of seats",
        }
    elif seats < len(attendee_ids):
        res['warning'] = {
            'title'   : "Warning: problems",
            'message' : "You need more seats for this session",
        }
    return res
```

8.3 Predefined osv.Model attributes for business objects

_constraints	list of tuples defining the Python constraints, in the form (func_name, message, fields).
_sql_constraints	list of tuples defining the SQL constraints, in the form (name, sql_def, message).

Exercise 4 - Add Python constraints

Add a constraint that checks that the instructor is not present in the attendees of his/her own session.

```
def _check_instructor_not_in_attendees(self, cr, uid, ids, context=None):
    for session in self.browse(cr, uid, ids, context):
        partners = [att.partner_id for att in session.attendee_ids]
        if session.instructor_id \
            and session.instructor_id in partners:
             return False
    return True

_columns = {
    ...
}

_constraints = [
    (_check_instructor_not_in_attendees,
     "The instructor can not be also an attendee!",
     ['instructor_id','attendee_ids']),
]
```

Exercise 5 - Add SQL constraints

With the help of PostgreSQL's doc http://www.postgresql.org/docs/8.4/static/ddl-constraints.html, add the following constraints:

1. *CHECK* that the course description and the course title are not the same
2. Make the Course's name *UNIQUE*
3. Make sure the Attendee table can not contain the same partner for the same session multiple times (*UNIQUE* on pairs)

```
class Course(osv.Model):
    _name = "openacademy.course"

    _columns = {
        ...
    }

    _sql_constraints = [

        ('name_description_check',
         'CHECK(name <> description)',
         'The title of the course should be different of the description'),

        ('name_unique',
         'UNIQUE(name)',
         'The title must be unique'),

    ]

(...)

class Attendee(osv.Model):
    _name = "openacademy.attendee"

    _columns = {
        ...
    }

    _sql_constraints = [
        ('partner_session_unique',
         'UNIQUE(partner_id, session_id)',
         'You can not insert the same attendee multiple times!'),
    ]
```

Exercise 6 - Add a duplicate option

Since we added a constraint for the Course name uniqueness, it is not possible to use the “duplicate” function anymore (Form > Duplicate). Re-implement your own “copy” method which allows to duplicate the Course object, changing the original name into “Copy of [original name]”.

Add the following function in the Course class:

```python
class Course(osv.Model):
    _name = "openacademy.course"

    # Allow to make a duplicate Course
    def copy(self, cr, uid, id, default, context=None):
        course = self.browse(cr, uid, id, context=context)
        new_name =  "Copy of %s" % course.name
        # =like is the original LIKE operator from SQL
        others_count = self.search(cr,  uid, [('name', '=like', new_name+'%')],
                                   count=True, context=context)
        if others_count > 0:
            new_name = "%s (%s)" % (new_name, others_count+1)
        default['name'] = new_name
        return super(Course, self).copy(cr, uid, id, default, context=context)

    _columns = {
        ...
    }
```

Exercise 7 - Active objects – Default values

Define the start_date default value as today. Add a field *active* in the class *Session*, and set the session as active by default.
See the class attribute _defaults in the technical memento.

In the openacademy file, in the Session class:

```python
class Session(osv.Model):
    _name = "openacademy.session"

    (...)

    _columns = {
        'name' : fields.char(string="Name", size=256, required=True),
        'start_date' : fields.date(string="Start date"),
        'duration' : fields.float(string="Duration", digits=(6,2),
                                  help="Duration in days"),
        'seats' : fields.integer(string="Number of seats"),
        # Is the record active in OpenERP? (visible)
        'active' : fields.boolean("Active"),

        # Function fields are calculated on-the-fly when reading records
        ...

        # Relational fields
        ...
    }

    _defaults = {
        # You can give a function or a lambda for default value. OpenERP will
        # automatically call it and get its return value at each record
        # creation. 'today' is a method of the object 'date' of module 'fields'
        # that returns -in the right timezone- the current date in the OpenERP
        # format
        'start_date' : fields.date.today,
        # Beware that is not the same as:
        # 'start_date' : fields.date.today(),
        # which actually call the method at OpenERP startup!
        'active' : True,
    }
```

Note:

Objects whose 'active' field is set to False *are not visible in OpenERP.*

9 Advanced Views

9.1 List & Tree

Lists include field elements, are created with type tree, and have a <tree> parent element.

Attributes	• colors: list of colors mapped to Python conditions • editable: top or bottom to allow in-place edit • toolbar: set to True to display the top level of object hierarchies as a side toolbar (example: the menu)
Allowed elements	field, group, separator, tree, button, filter, newline

```
<tree string="Idea Categories" toolbar="1" colors="blue:state==draft">
    <field name="name"/>
    <field name="state"/>
</tree>
```

Exercise 1 - List coloring

Modify the Session tree view in such a way that sessions lasting less than 5 days are colored blue, and the ones lasting more than 15 days are colored red.

Modify the Session tree view as follows:

```
<!-- session's tree/list view -->
<record model="ir.ui.view" id="session_tree_view">
  <field name="name">session.tree</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <!-- Colors are separated by semi-colons ';' and look like this:

         colorValue1:condition1;colorValue2:condition2;...

         Each color can be specified by using HTML color codes:
          - the hexadecimal form:
            #ff8080 => ff for red, 80 for green and blue
          - the color keywords:
            red, black, white, grey, ...

         When writing your condition, don't forget that you are
         in an XML file. You have to escape your special characters
         in order to make the whole file XML-compliant

         For example:
          - lesser than symbol is written &lt;
          - greater than symbol is written &gt; -->
    <tree string="Session Tree"
          colors="#0000ff:duration&lt;5;red:duration&gt;15">
      <field name="name"/>
      <field name="course_id"/>
      <!-- Since we use the duration field to determine the colors of each
           line, we have to include it in the view. If you don't want to
           make it visible, just hide it with the invisible attribute -->
      <field name="duration" invisible="1"/>
      <field name="taken_seats_percent" widget="progressbar"/>
    </tree>
  </field>
</record>
```

9.2 Calendars

Views used to display date fields as calendar events (<calendar> parent).

Attributes	• color: name of field for color segmentation • date_start: name of field containing event start date/time • date_stop: name of field containing event stop date/time
Allowed elements	field (to define the label for each calendar event)

```
<calendar string="Ideas" date_start="invent_date" color="inventor_id">
    <field name="name"/>
</calendar>
```

Exercise 2 - Calendar view

Add a Calendar view to the "Session" object enabling the user to view the events associated to the Open Academy.

1. Make a function field to convert duration (in days) to end_date

```
from openerp.osv import osv, fields

# Provide extended time manipulation tools
from datetime import datetime, timedelta

class Session(osv.Model):
    _name = "openacademy.session"

    ...

    def _determin_end_date(self, cr, uid, ids, field, arg, context=None):
        result = {}
        for session in self.browse(cr, uid, ids, context=context):
            if session.start_date and session.duration:
                # Get a datetime object from session.start_date
                start_date = datetime.strptime(session.start_date, "%Y-%m-%d")
                # Get a timedelta object from session.duration
                duration = timedelta( days=(session.duration - 1) )
                # Calculate the end time
                end_date = start_date + duration
                # Render in format YYYY-MM-DD
                result[session.id] = end_date.strftime("%Y-%m-%d")
            else:
                # Tip: In case session.start_date exists but session.duration
                #      not end_date will be equal to start_date
                result[session.id] = session.start_date
        return result

    # Beware that fnct_inv take a unique id and not a list of id
    def _set_end_date(self, cr, uid, id, field, value, arg, context=None):
        session = self.browse(cr, uid, id, context=context)
        if session.start_date and value:
            start_date = datetime.strptime(session.start_date, "%Y-%m-%d")
            end_date = datetime.strptime(value[:10], "%Y-%m-%d")
            duration = end_date - start_date
            self.write(cr, uid, id, {'duration' : (duration.days + 1)},
                       context=context)

    ...

    _columns = {
        ...

        # Function fields are calculated on-the-fly when reading records
        'taken_seats_percent' : fields.function(_taken_seats_percent,
            type='float', string='Taken Seats'),
        'end_date' : fields.function(_determin_end_date,
            fnct_inv=_set_end_date, type='date', string="End Date"),

        # Relational fields
        ...
    }
```

Note:

Because we made the inverse function, we are able to move sessions in the calendar view.

2. Add a calendar view to the Session object

```
<!-- calendar view -->
<record model="ir.ui.view" id="session_calendar_view">
  <field name="name">session.calendar</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <calendar string="Session Calendar"
              date_start="start_date"
              date_stop="end_date"
              color="instructor_id">
      <field name="name"/>
    </calendar>
  </field>
</record>
```

3. Update the action list of the Session view

```
<record model="ir.actions.act_window" id="session_list_action">
  <field name="name">Sessions</field>
  <field name="res_model">openacademy.session</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form,calendar</field>
</record>
```

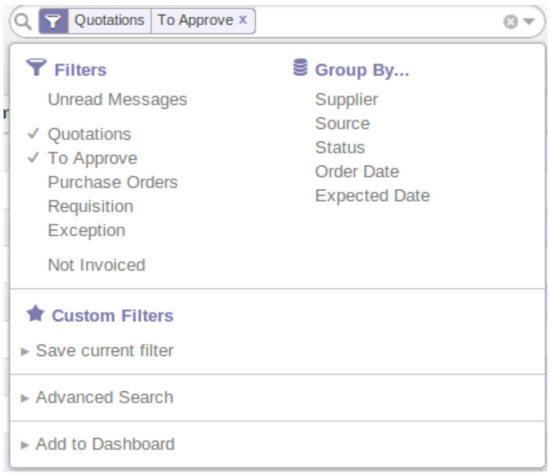
9.3 Search Views

Search views are used to customize the search panel on top of list views, and are declared with the search type, and a top-level <search> element.

After defining a search view with a unique id, add it to the action opening the list view using the search_view_id field in its declaration.

Allowed elements	field, group, separator, label, search, filter, newline, properties • filter elements allow defining button for domain filters • adding a context attribute to fields makes widgets that alter the search context (useful for context-sensitive fields, e.g. pricelist-dependent prices)

```
<search string="Ideas">
    <filter name="my_ideas" domain="[('inventor_id','=',uid)]"
        string="My Ideas" icon="terp-partner"/>
    <field name="name"/>
    <field name="description"/>
    <field name="inventor_id"/>
    <field name="country_id" widget="selection"/>
</search>
```



The action record that opens such a view may initialize search fields by its field *context*. The value of the field *context* is a Python dictionary that can modify the client's behavior. The keys of the dictionary are given a meaning depending on the following convention.

- The key `'default_foo'` initializes the field `'foo'` to the corresponding value in the form view.
- The key `'search_default_foo'` initializes the field `'foo'` to the corresponding value in the search view. Note that filter elements are like boolean fields.

Exercise 3 - Search views

Add a search view containing:

1. a field to search the courses based on their title and
2. a button to filter the courses of which the current user is the responsible. Make the latter selected by default.

1. Add the following view:

```
<record model="ir.ui.view" id="course_search_view">
    <field name="name">course.search</field>
    <field name="model">openacademy.course</field>
    <field name="arch" type="xml">
        <search string="Session Search">
```

```
            <filter string="My Courses" icon="terp-partner"
                    name="my_courses"
                    domain="[('responsible_id','=',uid)]"
                    help="My own sessions" />
            <field name="name"/>
        </search>
    </field>
</record>
```

2. Modify the action:

```
<record model="ir.actions.act_window" id="course_list_action">
  <field name="name">Courses</field>
  <field name="res_model">openacademy.course</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form</field>
  <!-- In case you have multiple search views, you can define which one you
       want to display

       <field name="search_view_id" ref="course_search_view"/>
  -->
  <field name="context">{'search_default_my_courses':1}</field>
  <field name="help" type="html">
    <p class="oe_view_nocontent_create">Create the first course</p>
  </field>
</record>
```

Note:

Check you have set the attribute select in your responsible_id field declaration to make sure you created an index to improve search performances.

9.4 Gantt Charts

Bar chart typically used to show project schedule (<gantt> parent element).

Attributes	same as <calendar>
Allowed elements	**field, level** • level elements are used to define the Gantt chart levels, with the enclosed field used as label for that drill-down level

```
<gantt string="Ideas" date_start="invent_date" color="inventor_id">
    <level object="idea.idea" link="id" domain="[]">
        <field name="inventor_id"/>
    </level>
</gantt>
```

Exercise 4 - Gantt charts

Add a Gantt Chart enabling the user to view the sessions scheduling linked to the Open Academy module. The sessions should be grouped by instructor.

1. Make a function field to transform duration (in days) to hours:

```
class Session(osv.Model):
    _name = "openacademy.session"

    (...)
    def _determin_hours_from_duration(self, cr, uid, ids, field,
                                      arg, context=None):
        result = {}
        sessions = self.browse(cr, uid, ids, context=context)
        for session in sessions:
            result[session.id] = (session.duration * 24 if session.duration
                                                    else 0)
        return result

    def _set_hours(self, cr, uid, id, field, value, arg, context=None):
        if value:
            self.write(cr, uid, id,
                       {'duration' : (value / 24)},
                       context=context)

    _columns = {
        ...

        # Function fields are calculated on-the-fly when reading records
        'taken_seats_percent' : fields.function(_taken_seats_percent,
            type='float', string='Taken Seats'),
        'end_date' : fields.function(_determin_end_date,
            fnct_inv=_set_end_date, type='date', string="End Date"),
        'hours' : fields.function(_determin_hours_from_duration,
            fnct_inv=_set_hours, type='float', string="Hours"),

        # Relational fields
        ...
    }
```

2. Add the following view:

```
<!-- gantt view -->
<record model="ir.ui.view" id="session_gantt_view">
  <field name="name">session.gantt</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <gantt string="Session Gantt" color="course_id"
           date_start="start_date" date_delay="hours">
      <level object="res.partner" link="instructor_id">
        <field name="name"/>
      </level>
    </gantt>
  </field>
</record>

...

<record model="ir.actions.act_window" id="session_list_action">
  <field name="name">Sessions</field>
  <field name="res_model">openacademy.session</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form,calendar,gantt</field>
</record>
```

9.5 (Charts) Graphs

Views used to display statistical charts (<graph> parent element).

Attributes	• type: type of chart: bar, pie (default) • orientation: horizontal, vertical
Allowed elements	**field, with specific behavior:** • first field in view is X axis, 2nd one is Y, 3rd one is Z • 2 fields required, 3rd one is optional • group attribute defines the GROUP BY field (set to 1) • operator attribute sets the aggregation operator to use for other fields when one field is grouped (+,*,**,min,max)

```
<graph string="Total idea score by Inventor" type="bar">
    <field name="inventor_id" />
    <field name="score" operator="+"/>
</graph>
```

Exercise 5 - Graph view

Add a Graph view **in the Session object** that displays, for each course, the number of attendees under the form of a bar chart.

1. First we need to add a functional field to the Session object:

```
class Session(osv.Model):
    _name = "openacademy.session"

    (...)

    _columns = {
        ...

        # Function fields are calculated on-the-fly when reading records
        'taken_seats_percent' : fields.function(_taken_seats_percent,
            type='float', string='Taken Seats'),
        'attendee_count': fields.function(_get_attendee_count,
            type='integer', string='Attendee Count', store=True),
        ...

        # Relational fields
        ...
    }
```

Note:

New in version 7.0
*You have to set the **store** flag of the function to True because graphs are computed from the database datas.*

2. Then of course the corresponding function:

```
def _get_attendee_count(self, cr, uid, ids, name, args, context=None):
    res = {}
    for session in self.browse(cr, uid, ids, context=context):
        res[session.id] = len(session.attendee_ids)
    return res
```

3. And finally create the view in the openacademy view file:

```
<!-- graph view -->
<record model="ir.ui.view" id="openacademy_session_graph_view">
  <field name="name">openacademy.session.graph</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <graph string="Participations by Courses" type="bar">
      <field name="course_id"/>
      <field name="attendee_count" operator="+"/>
    </graph>
  </field>
</record>

...

<record model="ir.actions.act_window" id="session_list_action">
  <field name="name">Sessions</field>
  <field name="res_model">openacademy.session</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form,calendar,gantt,graph</field>
</record>
```

Extra Exercise - Graph view in relation object

You can try to do the same for the *Course* object. Beware that the field sould be re-calculated when "attendee_ids" of the *Session* object is changed.

1. First we need to add a functional field to the Course object:

```
class Course(osv.Model):
    _name = "openacademy.course"

    (...)

    _columns = {
        'name' : fields.char(...),
        'description' : fields.text(...),

        # Function fields
        'attendee_count': fields.function(_get_attendee_count,
            type='integer', string='Attendee Count',
            store={
                'openacademy.session' :
                (_get_courses_from_sessions,['attendee_ids'],0)
            }),

        ...
    }
```

2. Then of course the corresponding function:

```
class Course(osv.Model):
    _name = "openacademy.course"

    def _get_attendee_count(self, cr, uid, ids, name, args, context=None):
        res = {}
        for course in self.browse(cr, uid, ids, context=context):
            res[course.id] = 0
            for session in course.session_ids:
                res[course.id] += len(session.attendee_ids)
        return res

    # /!\ This method is called from the session object!!
    def _get_courses_from_sessions(self, cr, uid, ids, context=None):
        sessions = self.browse(cr, uid, ids, context=context)
        # Return a list of Course ids with only one occurrence of each
        return list(set(sess.course_id.id for sess in sessions))

    _columns = {
        ...
    }
```

3. And finally create the view in the openacademy view file:

```
<!-- graph view -->
<record model="ir.ui.view" id="openacademy_course_graph_view">
  <field name="name">openacademy.course.graph</field>
  <field name="model">openacademy.course</field>
  <field name="arch" type="xml">
    <graph string="Participations by Courses" type="bar">
      <field name="name"/>
      <field name="attendee_count" operator="+"/>
    </graph>
  </field>
</record>

...

<record model="ir.actions.act_window" id="course_list_action">
  <field name="name">Courses</field>
  <field name="res_model">openacademy.course</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form,graph</field>
  <!-- In case you have multiple search views, you can define which one you
       want to display

       <field name="search_view_id" ref="course_search_view"/>
  -->
  <field name="context">{'search_default_my_courses':1}</field>
  <field name="help" type="html">
    <p class="oe_view_nocontent_create">Create the first course</p>
  </field>
</record>
```

9.6 Kanban Boards

Those views are available since OpenERP 6.1, and may be used to organize tasks, production processes, etc. A kanban view presents a set of columns of cards; each card represents a record, and columns represent the values of a given field. For instance, project tasks may be organized by stage (each column is a stage), or by responsible (each column is a user), and so on.

The following example is a simplification of the Kanban view of leads. The view is defined with *qweb* templates, and can mix form elements with HTML elements.

Exercise 6 - Kanban view

Add a Kanban view that displays sessions grouped by course (columns are thus courses).

1. Add the field “color” in your *Session* object

```
class Session(osv.Model):
    _name = "openacademy.session"

    ...
    _columns = {
        ...
        'color' : fields.integer('Color'),

        # Function fields are calculated on-the-fly when reading records
        ...

        # Relational fields
        ...
    }
```

2. Add the kanban view and update the action window

```
<!-- kanban view -->
<record model="ir.ui.view" id="view_openacad_session_kanban">
  <field name="name">openacad.session.kanban</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <kanban default_group_by="course_id">
      <field name="color"/>
      <templates>
        <t t-name="kanban-box">
          <div
          t-attf-class="oe_kanban_color_#{kanban_getcolor(record.color.raw_value)}
          oe_kanban_global_click_edit oe_semantic_html_override
          oe_kanban_card #{record.group_fancy==1 ? 'oe_kanban_card_fancy' :
          ''}">
            <div class="oe_dropdown_kanban">

              <!-- dropdown menu -->
              <div class="oe_dropdown_toggle">
                <span class="oe_e">#</span>
                <ul class="oe_dropdown_menu">
                  <li><a type="delete">Delete</a></li>
                  <li><ul class="oe_kanban_colorpicker"
                          data-field="color"/></li>
                </ul>
              </div>

              <div class="oe_clear"></div>
            </div>
            <div t-attf-class="oe_kanban_content">
              <!-- title -->
              Session name: <field name="name"/><br />
              Start date: <field name="start_date"/><br />
              duration: <field name="duration"/>
            </div>
        </t>
      </templates>
    </kanban>
  </field>
</record>

...

<record model="ir.actions.act_window" id="session_list_action">
  <field name="name">Sessions</field>
  <field name="res_model">openacademy.session</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form,calendar,gantt,graph,kanban</field>
</record>
```

10 Workflows

Workflows are models associated to business objects describing their dynamics. Workflows are also used to track processes that evolve over time.

Exercise 1 - Almost a workflow

Add a *state* field that will be used for defining a "workflow" on the object *Session*. A session can have three possible states: Draft (default), Confirmed and Done. In the session form, add a (read-only) field to visualize the state, and buttons to change it. The valid transitions are:

- Draft → Confirmed
- Confirmed → Draft
- Confirmed → Done
- Done → Draft

1. Add a new field in the dictionary "_columns" of the class Session in the file openacademy.py:

```
class Session(osv.Model):
    _name = "openacademy.session"

    ...

    _columns = {
        ...
        'state' : fields.selection([('draft','Draft'),
                                    ('confirmed','Confirmed'),
                                    ('done','Done')], string="State"),

        # Function fields are calculated on-the-fly when reading records
        ...

        # Relational fields
        ...
    }
```

2. Add a default value for the new field:

```
_defaults = {
    ...
    'active': True,
    'state': 'draft',
}
```

3. Define three new methods in the class Session. Those methods may be called to change a session's state field.

```
class Session(osv.Model):
    _name = "openacademy.session"

    def action_draft(self, cr, uid, ids, context=None):
        #set to "draft" state
        return self.write(cr, uid, ids, {'state':'draft'}, context=context)

    def action_confirm(self, cr, uid, ids, context=None):
        #set to "confirmed" state
        return self.write(cr, uid, ids, {'state':'confirmed'}, context=context)

    def action_done(self, cr, uid, ids, context=None):
        #set to "done" state
        return self.write(cr, uid, ids, {'state':'done'}, context=context)
```

4. Add three buttons in the Session form view, each one calling the corresponding method:

```
<!-- session’s form view -->
<record model="ir.ui.view" id="session_form_view">
  <field name="name">session.form</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <form string="Session Form" version="7.0">
      <header>
        <button name="action_draft" type="object" string="Reset to draft"
                states="confirmed,done" />
        <button name="action_confirm" type="object" string="Confirm"
                states="draft" class="oe_highlight" />
        <button name="action_done" type="object" string="Mark as done"
                states="confirmed" class="oe_highlight" />
        <field name="state" widget="statusbar" />
      </header>
      <sheet>
        <div class="oe_title">
          <label for="name" class="oe_edit_only" />
          <h1><field name="name" /></h1>
          <group colspan="2" col="2">
            <field name="course_id" placeholder="Course"/>
            <field name="instructor_id" placeholder="Instructor"/>
          </group>
        </div>
        <separator string="Schedule" />
        <group colspan="2" col="2">
          <field name="start_date" placeholder="Start Date"/>
          <field name="duration" placeholder="Duration"/>
          <field name="seats" placeholder="Seats"
                 on_change="onchange_taken_seats(seats, attendee_ids)"/>
        </group>
        <separator string="Attendees" />
        <field name="taken_seats_percent" widget="progressbar"/>
        <field name="attendee_ids"
               on_change="onchange_taken_seats(seats, attendee_ids)">
          <!-- ’editable’ attribute will set the position
               to push new elements in the list -->
          <tree string="" editable="bottom">
            <field name="partner_id"/>
          </tree>
        </field>
      </sheet>
    </form>
  </field>
</record>
```

A sales order generates an invoice and a shipping order is an example of workflow used in OpenERP.

Workflows may be associated with any object in OpenERP, and are entirely customizable. Workflows are used to structure and manage the lifecycles of business objects and documents, and define transitions, triggers, etc. with graphical tools. Workflows, activities (nodes or actions) and transitions (conditions) are declared as XML records, as usual. The tokens that navigate in workflows are called workitems.

Exercise 2 - Dynamic workflow editor

Using the workflow editor, create the same workflow as the one defined earlier for the Session object. Transform the Session form view such that the buttons change the state in the workflow.

Note:

A workflow associated to a session is created during the creation of that session. There is therefore no workflow instance associated to session instances created before the definition of the workflow.

1. Create the workflow using the web-client (Settings > Customization > Workflows > Workflows) and create a new workflow. Switch to the diagram view and add the nodes and transitions. A transition should be associated to the corresponding signal, and each activity (node) should call a function modifying the session state, according to the state in the workflow.
2. Modify the buttons previously created:

```
<!-- session's form view -->
<record model="ir.ui.view" id="session_form_view">
  <field name="name">session.form</field>
  <field name="model">openacademy.session</field>
  <field name="arch" type="xml">
    <form string="Session Form" version="7.0">
      <header>
        <button name="signal_draft" type="workflow"
                string="Reset to draft" states="confirmed,done" />
        <button name="signal_confirm" type="workflow" string="Confirm"
                states="draft" class="oe_highlight" />
        <button name="signal_done" type="workflow" string="Mark as done"
                states="confirmed" class="oe_highlight" />
        <field name="state" widget="statusbar" />
      </header>
      <sheet>
        ...
      </sheet>
    </form>
  </field>
</record>
```

3. If the function in the Draft activity is encoded, you can even remove the default state value in the Session class.

Note:

In order to check if instances of the workflow are indeed created when a session is created, you can go to the menu Session > Low Level Objects and look for the workflow. However, to be able to see that Low Level Objects menu, you need to add youself (Admin) to the Useability > No One group.

Exercise 3 - Automatic transitions

Add a transition Draft → Confirmed that is triggered automatically when the number of attendees in a session is more than half the number of seats of that session.

1. Add a transition between the activities Draft and Confirmed, without a signal, but instead with the condition:

```
taken_seats_percent > 50
```

Exercise 4 - Server actions

Create server actions and modify the previous workflow in order to re-create the same behaviour as previously, but without using the Python methods of the Session class.

11 Security

Access control mechanisms must be configured to achieve a coherent security policy.

11.1 Group-based access control mechanisms

Groups are created as normal records on the model “res.groups”, and granted menu access via menu definitions. However even without a menu, objects may still be accessible indirectly, so actual object-level permissions (read, write, create, unlink) must be defined for groups. They are usually inserted via CSV files inside modules. It is also possible to restrict access to specific fields on a view or object using the field’s groups attribute.

11.2 Access rights

Access rights are defined as records of the model “ir.model.access”. Each access right is associated to a model, a group (or no group for global access), and a set of permissions: read, write, create, unlink. Such access rights are usually created by a CSV file named after its model: “ir.model.access.csv”.

```
id,name,model_id/id,group_id/id,perm_read,perm_write,perm_create,perm_unlink
access_idea_idea,idea.idea,model_idea_idea,base.group_user,1,1,1,0
access_idea_vote,idea.vote,model_idea_vote,base.group_user,1,1,1,0
```

Exercise 1 - Add access control through the OpenERP interface

Create a new user “John Smith”. Then create a group “OpenAcademy / Session Read” with read access to the Session and Attendee objects.

1. Create a new user through the Settings > Users > Users menu.
2. Create a new group “session_read” using the Settings > Users > Groups menu with read rights on Session and Attendee objects.
3. Edit the user “John Smith” and add the group “session_read” to their groups (you may remove the others).
4. Log in as “John Smith” to check the access rights.

Exercise 2 - Add access control through data files in your module

Using an XML data file, create a group “OpenAcademy / Manager”, with no access rights defined yet (just create an empty group).

1. Create a new directory “security” in the “openacademy” directory. Create a new file “groups.xml”:

```
<?xml version="1.0" encoding="utf-8"?>
<openerp>
    <data>
        <record id="group_manager" model="res.groups">
            <field name="name">OpenAcademy / Manager</field>
        </record>
    </data>
</openerp>
```

2. Update “__openerp__.py” :

```
'update_xml': [
    # ...
    "security/groups.xml",
]
```

Exercise 3 - Add access control through data files in your module

Use a CSV file to add read, write, creation and deletion rights on the objects Course, Session and Attendees to the group *OpenAcademy / Manager*. You can also create rights associated to no group, such as a read only access on Course and a read only access on Session.

1. Define a new csv file "ir.model.access.csv" containing security rules:

```
id,name,model_id/id,group_id/id,perm_read,perm_write,perm_create,perm_unlink
course_full_manager,course_full_manager,model_openacademy_course,group_manager,1,1,1,1
course_read_all,course_read_all,model_openacademy_course,,1,0,0,0
session_full_all,session_full_all,model_openacademy_session,,1,1,1,1
attendee_full_all,attendee_full_all,model_openacademy_attendee,,1,1,1,1
```

2. Update "__openerp__.py" :

```
'update_xml': [
    # ...
    "security/ir.model.access.csv",
]
```

11.3 Record rules

A *record rule* restricts the access rights to a *subset* of records of the given model. A rule is a record of the model "ir.rule", and is associated to a model, a number of groups (many2many field), permissions to which the restriction applies, and a domain. The domain specifies to which records the access rights are limited.

Here is an example of a rule that prevents the deletion of leads that are not in state "cancel". Notice that the value of the field "groups" must follow the same convention as the method "write" of the ORM.

```
<record id="delete_cancelled_only" model="ir.rule">
    <field name="name">Only cancelled leads may be deleted</field>
    <field name="model_id" ref="crm.model_crm_lead"/>
    <field name="groups" eval="[(4, ref('base.group_sale_manager'))]"/>
    <field name="perm_read" eval="0"/>
    <field name="perm_write" eval="0"/>
    <field name="perm_create" eval="0"/>
    <field name="perm_unlink" eval="1" />
    <field name="domain_force">[('state','=','cancel')]</field>
</record>
```

Exercise 4 - Record rule

Add a record rule for the model Course and the group "OpenAcademy / Manager", that restricts "write" and "unlink" accesses to the responsible of a course. If a course has no responsible, all users of the group must be able to modify it.

1. Add the rule record in the file "openacademy/groups.xml":

```
<record id="only_responsible_can_modify" model="ir.rule">
    <field name="name">Only Responsible can modify Course</field>
    <field name="model_id" ref="model_openacademy_course"/>
    <field name="groups" eval="[(4, ref('openacademy.group_manager'))]"/>
    <field name="perm_read" eval="0"/>
    <field name="perm_write" eval="1"/>
    <field name="perm_create" eval="0"/>
    <field name="perm_unlink" eval="1"/>
    <field name="domain_force">
```

```
        ['|', ('responsible_id','=',False), ('responsible_id','=',user.id)]</field>
</record>
```

12 Wizards

12.1 Wizard objects (osv.TransientModel)

Wizards describe stateful interactive sessions with the user (or dialog boxes) through dynamic forms. A wizard is built simply by defining a model that extends the class “osv.TransientModel” instead of “osv.Model”. The class “osv.TransientModel” extends “osv.Model” and reuse all its existing mechanisms, with the following particularities:

- Wizard records are not meant to be persistent; they are automatically deleted from the database after a certain time. This is why they are called “transient”.
- Wizard models do not require explicit access rights: users have all permissions on wizard records.
- Wizard records may refer to regular records or wizard records through many2one fields, but regular records *cannot* refer to wizard records through a many2one field.

We want to create a wizard that allow users to create attendees for a particular session, or for a list of sessions at once. In a first step, the wizard will work for a single session.

Exercise 1 - Define the wizard class

Create a wizard model (inheriting from osv.TransientModel) with a many2one relationship with the Session object and a one2many relationship with an Attendee object (wizard object, too). The new Attendee object has a name field and a many2one relationship with the Partner object. Define the class *CreateAttendeeWizard* and implement its structure.

1. Create a new directory “wizard” in the module “openacademy”, and create a new file “wizard/create_attendee.py”.

```
from osv import osv,fields

class CreateAttendeeWizard(osv.TransientModel):
    _name = 'openacademy.create.attendee.wizard'
    _columns = {
        'session_id': fields.many2one('openacademy.session', 'Session',
                              required=True),
        'attendee_ids': fields.one2many('openacademy.attendee.wizard',
                              'wizard_id', 'Attendees'),
    }

class AttendeeWizard(osv.TransientModel):
    _name = 'openacademy.attendee.wizard'
    _columns = {
        'partner_id': fields.many2one('res.partner', 'Partner', required=True),
        'wizard_id':fields.many2one('openacademy.create.attendee.wizard'),
    }
```

2. Create an new file “wizard/__init__.py”.

```
import create_attendee
```

3. Update the “__init__.py” file under the “openacademy” module.

```
import wizard
```

12.2 Wizard execution

Wizards are launched by "ir.actions.act_window" records, with the field "target" set to value "new". The latter opens the wizard view into a popup window. The action is triggered by a menu item.

There is another way to launch the wizard: using an "ir.actions.act_window" record like above, but with an extra field "src_model" that specifies in the context of which model the action is available. The wizard will appear in the contextual actions of the model, on the right-hand side bar. Because of some internal hooks in the ORM, such an action is declared in XML with the tag "act_window".

```
<act_window id="session_create_attendee_wizard"
    name="Add Attendees"
    src_model="context_model_name"
    res_model="wizard_model_name"
    view_mode="form"
    target="new"
    key2="client_action_multi"/>
```

Note:

The field "key2" defines a kind of action category. Its possible values are: "client_action_multi" (typically for wizards), "client_print_multi" (typically for reports), and "client_action_relate" (typically for related views).

Exercise 2 - Make the wizard available through a menuitem

Create a menuitem and the necessary action to use the wizard.

1. Create a new file "wizard/create_attendee_view.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<openerp>
    <data>
        <record model="ir.actions.act_window" id="create_attendee_wizard_action">
            <field name="name">Add attendee</field>
            <field name="res_model">openacademy.create.attendee.wizard</field>
            <field name="view_type">form</field>
            <field name="view_mode">form</field>
            <field name="target">new</field>
        </record>

        <menuitem name="Add attendee" parent="openacademy_menu"
            id="create_attendee_wizard_menu"
            action="create_attendee_wizard_action"/>
    </data>
</openerp>
```

2. Update "__openerp__.py" under the main "openacademy" project :

```
'data': [
    # ...
    'wizard/create_attendee_view.xml',
],
```

12.3 Wizard views

Wizards use regular views and their buttons may use the attribute *special="cancel"* to close the wizard window without saving.

```
<button string="Do it!" type="object" name="some_action"/>
<button string="Cancel" special="cancel"/>
```

Exercise 3 - Customise the form view

Customise the form view in order to show all the fields of the class.

Add the view in the file “wizard/create_attendee_view.xml”:

```
<form string="Add attendee" col="4" version="7.0">
    <group colspan="2">
    <field name="session_id"  colspan="2"/>
    </group>

    <field name="attendee_ids" nolabel="1" colspan="4">
        <tree string="Attendees" editable="bottom">
            <field name="partner_id"/>
        </tree>
    </field>
    <footer>
    <button type="special" special="cancel"
            string="Cancel" icon="gtk-cancel"/>
    <button type="object" name="action_add_attendee"
            string="Add attendees" icon="gtk-ok"
            confirm="Are you sure you want to add those attendees?"/>
    </footer>
</form>
```

Exercise 4 - Create methods

Create the method *action_add_attendee* in your class *CreateAttendeeWizard*, implement it, and add a button in the form view to call it. Add also a button “Cancel” that closes the wizard window.

1. Create the method *action_add_attendee* in the class *CreateAttendeeWizard*.

```
def action_add_attendee(self, cr, uid, ids, context=None):
    attendee_model = self.pool.get('openacademy.attendee')
    wizard = self.browse(cr, uid, ids[0], context=context)
    for attendee in wizard.attendee_ids:
        attendee_model.create(cr, uid, {
            'partner_id': attendee.partner_id.id,
            'session_id': wizard.session_id.id,
        })
    return {}
```

Note:

In order to close the window at the end of the method, you can either return the empty dictionary {}*, or explicitly return a client action to close the window* {‘type’: ‘ir.actions.act_window.close’}.

2. Add the buttons to the form view.

```
<record model="ir.ui.view" id="create_attendee_form_view">
    ...
    <field name="arch" type="xml">
        <form string="Add attendee" col="2">
            ...
            <button string="Cancel" icon="gtk-cancel"
                special="cancel"/>
            <button string="Add attendees" icon="gtk-ok"
```

```
                type="object" name="action_add_attendee"/>
        </form>
    </field>
</record>
```

Note:

Creating attendees this way does not trigger the workflow of sessions. This is because the session record is not explicitly "touched". You should either "wake up" the session record explicitly, or write on the session record.

1. Improve the method *action_add_attendee* such that it writes on the session:

```
def action_add_attendee(self, cr, uid, ids, context=None):
    session_model = self.pool.get('openacademy.session')
    wizard = self.browse(cr, uid, ids[0], context=context)
    session_ids = [wizard.session_id.id]
    att_data = [{'partner_id': att.partner_id.id} for att in wizard.attendee_ids]
    session_model.write(cr, uid, session_ids,
        {'attendee_ids': [(0, 0, data) for data in att_data]},
        context)
    return {}
```

Exercise 5 - Bind the wizard to the context bar

Bind the wizard to the context bar of the session model.
Hint: use the argument "context" to define the current session as default value for the field "session_id" in the wizard.

1. Add the required XML element in the file "wizard/create_attendee_view.xml":

```
<act_window id="session_create_attendee_wizard"
    name="Add Attendees"
    src_model="openacademy.session"
    res_model="openacademy.create.attendee.wizard"
    view_mode="form"
    target="new"
    key2="client_action_multi"/>
```

2. In your class *CreateAttendeeWizard*, define a default value for the field "session_id":

```
class CreateAttendeeWizard(osv.TransientModel):
    # ...

    def _get_active_session(self, cr, uid, context):
        if context.get('active_model') == 'openacademy.session':
            return context.get('active_id', False)
        return False

    _defaults = {
        'session_id': _get_active_session,
    }
```

Extra Exercise - Wizard on multiple records

Make the wizard able to add attendees to several sessions at once.

1. Change the model of the wizard, replacing the many2one field "session_id" by a many2many field "session_ids". Adapt the methods accordingly.

```python
class CreateAttendeeWizard(osv.TransientModel):
    _name = 'openacademy.create.attendee.wizard'
    _columns = {
        'session_ids': fields.many2many('openacademy.session', string='Session',
                             required=True),
        'attendee_ids': fields.one2many('openacademy.attendee.wizard',
                             'wizard_id', 'Attendees'),
    }

    def _get_active_sessions(self, cr, uid, context):
        if context.get('active_model') == 'openacademy.session':
            return context.get('active_ids', False)
        return False

    _defaults = {
        'session_ids': _get_active_sessions,
    }

    def action_add_attendee(self, cr, uid, ids, context=None):
        session_model = self.pool.get('openacademy.session')
        wizard = self.browse(cr, uid, ids[0], context=context)
        session_ids = [sess.id for sess in wizard.session_ids]
        att_data = [{'partner_id':att.partner_id.id} for att in wizard.attendee_ids]
        session_model.write(cr, uid, session_ids,
            {'attendee_ids': [(0, 0, data) for data in att_data]},
            context)
        return {}
```

2. Adapt the form view of the wizard.

```xml
<record model="ir.ui.view" id="create_attendee_form_view">
    <field name="name">openacademy.create.attendee.wizard.form</field>
    <field name="model">openacademy.create.attendee.wizard</field>
    <field name="type">form</field>
    <field name="arch" type="xml">
        <form string="Add attendee" col="4" version="7.0">
                <group colspan="2">
                <field name="session_ids"  colspan="2"/>
                </group>

                <field name="attendee_ids" nolabel="1" colspan="4">
                    <tree string="Attendees" editable="bottom">
                        <field name="partner_id"/>
                    </tree>
                </field>
                <footer>
                <button type="special" special="cancel"
                        string="Cancel" icon="gtk-cancel"/>
                <button type="object" name="action_add_attendee"
                        string="Add attendees" icon="gtk-ok"
                        confirm="Are you sure you want to add those attendees?"/>
                </footer>
            </form>
    </field>
</record>
```

13 Internationalization

Each module can provide its own translations within the i18n directory, by having files named LANG.po where LANG is the locale code for the language, or the language and country combination when they differ (e.g. pt.po or pt_BR.po). Translations will be loaded automatically by OpenERP for all enabled languages. Developers always use English when creating a module, then export the module terms using OpenERP's gettext POT export feature (Settings>Translations>Export a Translation File without specifying a language), to create the module template POT file, and then derive the translated PO files. Many IDE's have plugins or modes for editing and merging PO/POT files.

Tip:

The GNU gettext format (Portable Object) used by OpenERP is integrated into LaunchPad, making it an online collaborative translation platform.

```
|- idea/               # The module directory
  |- i18n/             # Translation files
    | - idea.pot       # Translation Template (exported from OpenERP)
    | - fr.po          # French translation
    | - pt_BR.po       # Brazilian Portuguese translation
    | (...)
```

Tip:

By default OpenERP's POT export only extracts labels inside XML files or inside field definitions in Python code, but any Python string can be translated this way by surrounding it with the tools.translate._ method (e.g. _('Label'))

Exercise 1 - Translate a module

Choose a second language for your OpenERP installation. Translate your module using the facilities provided by OpenERP.

1. Create a "i18n" directory in the OpenAcademy folder.
2. Install the target language in OpenERP (Administration > Translations > Load an Official Translation)
3. Synchronise the terms to translate through Administration > Translations > Application Terms > Synchronize Translations
4. Create the template translation file "openacademy.pot" by exporting (Administration > Translations > Import/Export > Export Translation) without specifying a language, and save it in the i18n directory.
5. Create the French (or any other language) translation file "fr_FR.po" by exporting (Administration > Translations > Import/Export > Export Translation) and specifying the French language, and save it in the i18n directory.
6. Open "fr_FR.po" with *poedit* (or a simple text editor) and translates the terms.

Note:

By default OpenERP's POT export only extracts labels inside XML records or inside field definitions in Python code, but any Python string can be translated by surrounding it with the tools.translate._ method (e.g. _('Label'))

7. Update the "openacademy.py".

```
from tools.translate import _
```

8. Add the "_" operator where it is needed, then re-do steps 3, 4, 5 and 6.

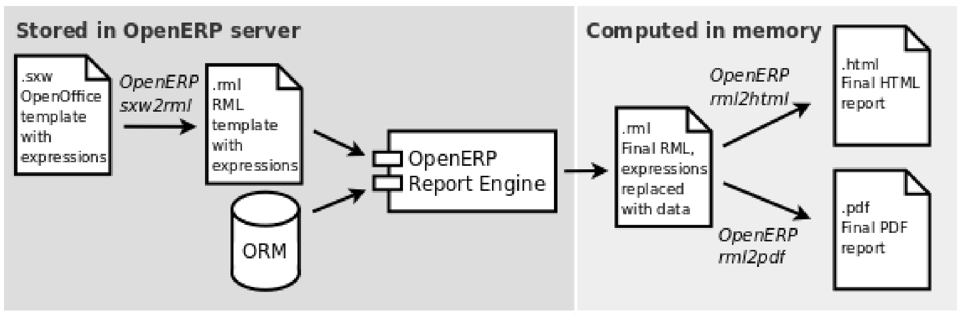
14 Reporting

14.1 Printed reports

Reports in OpenERP can be rendered in different ways:

- **Custom reports**: those reports can be directly created via the client interface, no programming required. Those reports are represented by business objects (ir.report.custom)
- **High quality personalized reports using openreport**: no programming required but you have to write 2 small XML files:
  - a template which indicates the data you plan to report
  - an XSL : RML stylesheet
- **High quality reports using the WebKit engine**: based on Mako HTML templates (http://www.makotemplates.org) and a tool to convert the result to PDF (wkhtmltopdf). This reporting engine is a contribution from Camptocamp (http://www.camptocamp.com).
- **Hard coded reports**
- **OpenOffice Writer templates**

There are several report engines in OpenERP, to produce reports from different sources and in many formats.



Expressions used in OpenERP report templates

[[<content>]]	double brackets content is evaluated as a Python expression based on the following expressions

Predefined expressions :

- *objects* contains the list of records to print
- *data* comes from the wizard launching the report
- *user* contains the current user (as per browse())
- *time* gives access to Python time module
- *repeatIn(list,'var','tag')* repeats the current parent element named tag for each object in list, making the object available as var during each loop
- *setTag('tag1','tag2')* replaces the parent RML tag1 with tag2
- *removeParentNode('tag')* removes parent RML element tag
- *formatLang(value, digits=2, date=False, date_time=False, grouping=True, monetary=False)* can be used to format a date, time or amount according to the locale
- *setLang('lang_code')* sets the current language and locale for translations

RML Reports

Exercise 1 - Install the OpenOffice plugin

Install the report designer module and add the OpenOffice OpenERP plugin.

Install the base > report designer module. Then go to Administration > Customization > Reporting > Report Designer then save the zip file

Exercise 2 - Create a new report

Create a report for the Session object, displaying for each session its name, date, duration, percentage of completion, responsible name and list of attendees.

```
[[repeatIn(objects,'session')]]
[[ setLang(session.course_id.responsible_id.context_lang) ]]

Name: [[ session.name ]]
Date: [[ formatLang(session.date,date=True) ]]
Duration: [[ formatLang(session.duration,digits=2) ]]
Percentage of completion: [[ session.perc_completion, digits=2 ]]
Responsable: [[ session.responsible_id.name ]]

=============================================    ====================================
Partner                                          Signature
=============================================    ====================================
[[repeatIn(session.attendee_ids,'attendee')]]
-------------------------------------------------------------------------------------
[[ attendee.partner_id ]]
=============================================    ====================================
```

Exercise 3 - RML

Export the OpenOffice report to a RML file, add it to your module and add an action to the session object for printing the report.

1. Create a "report" folder in your module and save the rml file into it.
2. Add an xml file for report declaration in this folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<openerp>
    <data>
        <report id="session_report" string="Print Sessions"
            model="openacademy.session" name="session.report"
            rml="openacademy/report/session.rml" />
    </data>
</openerp>
```

3. Update the __openerp__.py file to add a reference to this declaration file.

Exercise 4 - RML

Improve your report. Prevent the "start date" line from appearing in the report if no value is associated to it, and display the percentage of taken seats in red if it is lower than 10.

1. Remove "start date" line

```
<para style="Standard">Duration:
    [[ session.duration and formatLang(session.duration,digits=2) or
        (removeParentNode('para')) ]]
</para>
```

2. Set the font color on the percentage of taken seats.

```
<para style="Standard">Remaining seats percentage:
    <font color="black">
        [[session.taken_seats_percent&lt;50 and setTag('font','font',{'color':'red'})]]
        [[session.taken_seats_percent]]
    </font>
</para>
```

WebKit reports

To generate such reports, install the module *report_webkit*. A report is made from a Mako template file, and the report action is declared in XML as

```
<report string="WebKit invoice"
    id="report_webkit_html"
    model="account.invoice"
    name="webkitaccount.invoice"
    file="report_webkit_sample/report/report_webkit_html.mako"
    webkit_header="report_webkit.ir_header_webkit_basesample0"
    report_type="webkit"
    auto="False"/>
```

This example is taken from the module *report_webkit_sample*. Open and read the file report/report_webkit_html.mako in that module to see what a Mako template looks like.

Exercise 5 - Create a WebKit report

Create a report for the Session object, displaying for each session its name, date, duration, percentage of completion, responsible name and list of attendees.

1. In the module openacademy, create the file report/session_report.mako:

```
<html>
<head>
    <style type="text/css">${css}</style>
</head>
<body>
    <h1>Session Report</h1>
    % for session in objects:
        <h2>${session.course_id.name} - ${session.name}</h2>
        <p>From ${formatLang(session.start_date, date=True)} to
            ${formatLang(session.stop_date, date=True)}.</p>
        <p>Attendees:
            <ul>
                % for att in session.attendee_ids:
                    <li>${att.partner_id.name}</li>
                % endfor
            </ul>
        </p>
    % endfor
</body>
</html>
```

Note that it is possible to translate the static terms in the report by using the function "_" in expressions, like `${_('Session Report')}`.

2. Create the data file report/openacademy.xml with the report action:

```
<?xml version="1.0" encoding="UTF-8"?>
<openerp>
    <data>
        <report id="report_webkit"
            model="openacademy.session"
            name="openacademy.session.report"
            file="openacademy/report/session_report.mako"
            string="Session Report"
            report_type="webkit"/>
    </data>
</openerp>
```

3. Add the former file in the data section of __openerp__.py:

```
'data': [
    ...
    'report/openacademy.xml',
],
```

14.2 Dashboards

Exercise 6 - Define a Dashboard

Define a dashboard containing the graph view you created, the sessions calendar view and a list view of the courses (switchable to a form view). This dashboard should be available through a menuitem in the menu, and automatically displayed in the web client when the OpenAcademy main menu is selected.

1. Create a board_session_view.xml in your module. It should contain the board view, the actions referenced in that view, an action to open the dashboard, as well as a re-definition of the main menu item "openacademy_menu" to add the action to open the dashboard.

```
<?xml version="1.0"?>
<openerp>
     <data>

          <record model="ir.actions.act_window" id="act_session_graph">
               <field name="res_model">openacademy.session</field>
               <field name="view_type">form</field>
               <field name="view_mode">graph</field>
               <field name="view_id" ref="openacademy.openacademy_session_graph_view"/>
          </record>
          <record model="ir.actions.act_window" id="act_session_calendar">
               <field name="res_model">openacademy.session</field>
               <field name="view_type">form</field>
               <field name="view_mode">calendar</field>
               <field name="view_id" ref="openacademy.session_calendar_view"/>
          </record>
          <record model="ir.actions.act_window" id="act_course_list">
               <field name="res_model">openacademy.course</field>
               <field name="view_type">form</field>
               <field name="view_mode">tree,form</field>
          </record>

          <record model="ir.ui.view" id="board_session_form">
               <field name="name">Session Dashboard Form</field>
               <field name="model">board.board</field>
               <field name="type">form</field>
               <field name="arch" type="xml">
```

```
                    <form string="Session Dashboard" version="7.0">
                         <board style="2-1">
                            <column>
                                   <action
                                        string="Attendees by course"
                                        name="%(act_session_graph)d"
                                        colspan="4"
                                        height="150"
                                        width="510"/>
                                    <action
                                        string="Sessions"
                                        name="%(act_session_calendar)d"
                                        colspan="4"/>
                              </column>
                              <column>
                                     <action
                                        string="Courses"
                                        name="%(act_course_list)d"
                                        colspan="4"/>
                              </column>
                    </board>
                </form>
            </field>
        </record>

        <record model="ir.actions.act_window" id="open_board_session">
            <field name="name">Session Dashboard</field>
            <field name="res_model">board.board</field>
            <field name="view_type">form</field>
            <field name="view_mode">form</field>
            <field name="usage">menu</field>
            <field name="view_id" ref="board_session_form"/>
        </record>

        <menuitem id="openacademy_menu" name="OpenAcademy" action="open_board_session"/>

        <menuitem id="board.menu_dashboard" name="Dashboard" sequence="0" parent="openacademy_all

        <menuitem
            name="Session Dashboard" parent="board.menu_dashboard"
            action="open_board_session"
            sequence="1"
            id="menu_board_session" icon="terp-graph"
            />

    </data>
</openerp>
```

Note:

The available styles are “1”, “1-1”, “2-1”, “1-2”, “1-1-1”. The numbers represent the colspan of each column.

2. Update your __openerp__.py file to reference your new view file.

15 WebServices

The web-service module offer a common interface for all web-services :

- SOAP

- XML-RPC
- NET-RPC

Business objects can also be accessed via the distributed object mechanism. They can all be modified via the client interface with contextual views.

OpenERP is accessible through XML-RPC interfaces, for which libraries exist in many languages.

15.1 XML-RPC Library

The following example is a Python program that interacts with an OpenERP server with the library xmlrpclib.

```
import xmlrpclib
# ... define HOST, PORT, DB, USER, PASS
url = 'http://%s:%d/xmlrpc/common' % (HOST,PORT)
sock = xmlrpclib.ServerProxy(url)
uid = sock.login(DB,USER,PASS)
print "Logged in as %s (uid:%d)" % (USER,uid)

# Create a new idea
url = 'http://%s:%d/xmlrpc/object' % (HOST,PORT)
sock = xmlrpclib.ServerProxy(url)
args = {
    'name' : 'Another idea',
    'description' : 'This is another idea of mine',
    'inventor_id': uid,
}
idea_id = sock.execute(DB,uid,PASS,'idea.idea','create',args)
```

Exercise 1 - Add a new service to the client

Write a Python program able to send *XML-RPC* requests to a PC running OpenERP (yours, or your instructor's). This program should display all the sessions, and their corresponding number of seats. It should also create a new session for one of the courses.

1. Create a new repository "xml-rpc" under the "openacademy" repository. Under it, create a new file "test.py" :

```
import xmlrpclib

HOST='192.168.0.44'
PORT=8069
DB='openacademy'
USER='admin'
PASS='admin'

url = 'http://%s:%d/xmlrpc/' % (HOST,PORT)
common_proxy = xmlrpclib.ServerProxy(url+'common')
object_proxy = xmlrpclib.ServerProxy(url+'object')

def execute(\*args):
        return object_proxy.execute(DB,uid,PASS,*args)

# 1. Login
uid = common_proxy.login(DB,USER,PASS)
print "Logged in as %s (uid:%d)" % (USER,uid)

# 2. Read the sessions
session_ids = execute('openacademy.session','search',[])
sessions = execute('openacademy.session','read',session_ids, ['name','seats'])
```

```
for session in sessions :
    print "Session name :%s (%s seats)" % (session['name'], session['seats'])

# 3.create a new session
session_id = execute('openacademy.session', 'create',
                     {'name' : 'My session',
                      'course_id' : 2, })
```

2. Instead of adding the session to a course with a known id, we can search for a course based on its name.

```
# 3.create a new session for the "Functional" course
course_id = execute('openacademy.course', 'search', [('name','ilike','Functional')])[0]
session_id = execute('openacademy.session', 'create',
                     {'name' : 'My session',
                      'course_id' : course_id, })
```

Note:

*Through XML-RPC, it is possible to create an object (*create*), search objects (*search*, used with , read data, update data and delete objects (*unlink*) an object through*

15.2 OpenERP Client Library

The OpenERP Client Library (http://pypi.python.org/pypi/openerp-client-lib) is a Python library to communicate with an OpenERP server using its web services in an user-friendly way. It provides simple wrapper objects to abstract the bare XML-RPC calls.

If necessary, install the library:

```
$ sudo easy_install openerp-client-lib
```

The following Python program is equivalent to the example above.

```
import openerplib
# ... define HOST, PORT, DB, USER, PASS
connection = openerplib.get_connection(hostname=HOST, port=PORT, database=DB,
                            login=USER, password=PASS)
connection.check_login()
print "Logged in as %s (uid:%d)" % (connection.login, connection.user_id)

# create an idea
idea_model = connection.get_model('idea.idea')
values = {
    'name': 'Another idea',
    'description': 'This is another idea of mine',
    'inventor_id': connection.user_id,
}
idea_id = idea_model.create(values)
```

Exercise 2 - Add a new service to the client

Do the same as Exercise 1, but this time using `openerplib`.

1. In the new repository “xml-rpc”, create a new file “test-openerplib.py” :

```
import openerplib

HOST='192.168.0.44'
PORT=8069
DB='openacademy'
```

```
USER='admin'
PASS='admin'
connection = openerplib.get_connection(hostname=HOST, port=PORT, database=DB,
                            login=USER, password=PASS)

# 1. Login
connection.check_login()
print "Logged in as %s (uid:%d)" % (connection.login, connection.user_id)

# 2. Read the sessions
session_model = connection.get_model('openacademy.session')
session_ids = session_model.search([])
sessions = session_model.read(session_ids, ['name', 'seats'])
for session in sessions:
    print "Session name :%s (%s seats)" % (session['name'], session['seats'])

# 3.create a new session
course_model = connection.get_model('openacademy.course')
course_id = course_model.search([('name','ilike','Functional')])[0]
session_id = session_model.create({'name': 'My session', 'course_id': course_id})
```