Odoo Python Bootcamp

Day Odoo — Modules, Models and Magic

May 12, 2025

First Contact with Odoo Code

Welcome ! Today we stop writing code and scripting in a vacuum and step into the world of Odoo. You know it very well already ofc, but let's look at the technical point of view. We'll learn what makes Odoo tick, how to read a module, and then begin writing our own based on an example.

Part 0 — Installing Odoo Locally (for Development)

Before you can develop modules, you need a local Odoo instance running on your machine. This section walks you through the setup from scratch.

0.1 — Why use a virtual environment?

As we've succintly seen last week with the Spotipy app, Python virtual environments allow us to:

- Isolate project dependencies (no risk of breaking system packages)
- Keep things clean, especially when working on multiple projects
- Easily install and remove packages without admin rights

We'll use a venv to install and run Odoo locally, without touching the global Python environment.

0.2 — System Requirements

You should already have:

- **Python 3.10+** (run `python3 -version` to check)
- **PostgreSQL 15+** (run `psql -version` to check)
- **git**, **make**, **pip**

If you're missing any of these, now you know how to fix that hopefully !

0.3 — Step-by-step Installation

Step 1 — Clone the Odoo Source Code

```
git clone https://github.com/odoo/odoo.git -b 18.0 --depth 1
cd odoo
```

Step 2 — Create and activate a virtual environment

```
python3 -m venv venv
source venv/bin/activate
```

Your terminal should now start with `(venv)` *— this means the virtual environment is active.*

Step 3 — Upgrade pip and install dependencies

```
pip install --upgrade pip
pip install -r requirements.txt
```

If you see warnings, read them — errors mean something went wrong. Try understanding them, don't hesitate to call for help if somethin gis really not clear.

Step 4 — Create a PostgreSQL user

If you don't already have one, create a PostgreSQL user called `odoo`:

```
sudo -u postgres createuser -s odoo
```

Step 5 — Start Odoo

This is a command line with some options. How would you add an option that would prevent the worker from being killed during the debugging process? it will be useful, try to improve this CLI to add it !

```
./odoo-bin -d test_db -i real_estate --addons-path=addons --dev=all
```

Check on https://www.odoo.com/documentation/18.0/developer/tutorials/setup_guide.html
Some Then visit `http://localhost:8069` in your browser.

0.4 — Your Mission

By the end of this section, you should:

- Have Odoo 18 running on `localhost:8069`
- Be able to log in, create a new database (or use one of your own !), and enable Developer Mode and know what it does (rouglhy).
- Be ready to create and test your own module (again... roughly :D)

Part 1 — Odoo Architecture Crash Course

Core concepts:

- Odoo is built in Python
- Uses PostgreSQL as its database
- Employs an ORM (Object-Relational Mapping) layer
- UI is defined in XML + rendered in JS
- Entire system is modular. Brings a lot of complexity, but also what makes odoo so powerful !

Here's a minimal module layout:

```
my_module/
|__ __manifest__.py
|__ __init__.py
|__ models/
|    |__ __init__.py
|    |__ my_model.py
|__ views/
|    |__ my_model_views.xml
|__ security/
|    |__ ir.model.access.csv
```

Part 2 — Real Example: `real_estate` Module

You've been given access to the `real_estate` module.

Goals:

- Understand the structure and logic of a complete module
- Populate the system with realistic property and offer data
- Identify potential bugs or logic flaws (in code, views, or logic flows)

Tasks: Explore the module. Here are some questions to guide your exploration; you don't have to answer to everything, it is just here to help you:

- How is the model `estate.property` defined? Which fields does it inherit and which are added?
- Which views are defined? (list, form, kanban, etc.)
- What fields use `@api.onchange` or `@api.depends`? What's the difference between the two?
- What is the default state of a new property? What triggers a state change?
- Are any buttons hidden depending on the state? Try every button and describe its effect.

- Create properties with edge-case data (e.g. no name, selling price = 0, negative values). What breaks?
- What happens if you try to accept multiple offers on the same property?
- What does the method `action_sold()` do? What does it not do that it maybe should?
- Are any computed fields editable? Should they be?
- What happens if you delete an accepted offer? Try it.
- Are the access rights (ir.model.access.csv) too permissive or too strict?
- Is the Kanban view showing useful info? What would you add or change?
- How would you improve the UX of the form view?
- Is there validation to prevent setting a selling price higher than expected price?
- Can you create offers for archived properties? Should you be able to?
- Is the chatter being used meaningfully? Which messages are automated, which are manual?

To do:

- Propose at least 3 improvements to the module — either functional or technical
- Find a bug or inconsistency and describe it clearly with steps to reproduce
- Create your own property, make several offers, and take it through the full lifecycle: new → offer received → offer accepted → sold
- Suggest a computed field that would add value (e.g. "profit margin" or "days on market") and describe how to implement it

Part X — Odoo Shell Exercises (With real_estate)

Welcome to the Odoo Shell — your playground for inspecting data, modifying records, and writing quick automation scripts in a live Odoo environment.

How to launch the shell: Use the same command as before, but adding additionally the option shell

```
./odoo-bin shell -d your_db_name <your other options...>
```

You're now inside a live Python console with access to all your models via the ‘env‘ object.

Let's practice with the 'real_estate' module you already know.

Shell & Python Basics

- Print the number of properties with a garage
- List the names of properties that have no tag
- Set the garden area to 0 for all properties that have no garden

Business Logic & Automation

- Cancel all properties with an expected price under 100,000 and no offers
- Auto-generate a unique reference for each property
- For each partner, print how many offers they've made
- Cancel any property still in “new” state for more than 60 days
- Create a property, assign it to the current user, and attach a tag
- Post a message in the chatter of all properties with no offers

Challenge — Real-World Scenarios

- Mark properties as "sold" if their accepted offer has a selling price higher than expected
- Archive all partners who never made any offers
- Generate a summary of the number of properties per state (as a dictionary)
- Add a chatter message on properties with total area over 300m^2
- Assign buyer_id from the accepted offer if missing
- Create a random offer for properties that have none
- Assign a default tag to new properties that are missing one
- Post a warning if selling_price is set but buyer_id is empty

- Archive all refused offers older than 60 days
- Automatically mark as "sold" any property with both buyer and selling price set
- Add a log note to properties that have received more than 2 offers
- Refuse all offers where the price is less than 50% of expected_price
- Count how many properties were sold this month
- Increase expected_price by 5% for all active properties in “new” state

Bonus challenge: Pick one of the above and turn it into a button action or server action inside the real_estate module.

Advanced Shell Work

These exercises are more ambitious — they require combining multiple steps, writing functions, and thinking like a developer. Just like you did the other days ! You’ll likely need to use:

- multiple model queries
- filtering logic with loops and conditions
- Python dictionaries and lists
- sorting or grouping
- at least one function per script

1. Partner Portfolio Report

Goal: For each partner, generate a full report of:

- how many offers they’ve made
- how many were accepted
- the average offer value
- the total value of properties they’ve bought

Bonus: sort the report by total value descending.
—

2. Detect and Fix Orphaned Data

Goal: Write a script to find and clean up “orphaned” records:

- Offers with no related property
- Properties in “sold” state with no buyer
- Buyers who are not marked active

—

3. Sales Performance Analyzer

Goal: For each salesperson:

- Count how many properties they've sold
- Calculate the average margin (selling vs expected price)
- Calculate the average delay (days between property creation and sale)

Bonus: Format the results in a leaderboard style output. Your choice of style, make it shinee (or not, it's okay)

—

4. Smart Property Promotion Engine

Goal: Find all "new" properties that:

- Have not received an offer in the last 30 days
- Have a high rating or desirable tags (e.g. "Luxury")
- Are located in a specific location (if field exists, if not, create it ! countries could be intereseting)

Then:

- Reduce their expected price by 5%
- Add a "Promo" tag
- Post a promotion message in the chatter

—

Part X + 1 — Debug This!

Here are some broken or suspicious Odoo code snippets. Your job:

- Understand what the code is trying to do
- Identify and fix the bug or anti-pattern
- Add a comment in the code explaining what was wrong

You can test them inside your training modules (e.g., `real_estate` or `training.course`).

Example 1 — Misused `@api.depends`

```
@api.depends('start_date')
def _compute_duration(self):
    for rec in self:
        rec.duration = rec.end_date - rec.start_date
```

Hint: It doesn't update when `end_date` changes.

Example 2 — Broken Constraint

```
@api.constrains('expected_price', 'selling_price')
def _check_prices(self):
    for rec in self:
        if rec.expected_price > rec.selling_price:
            raise ValidationError("Selling␣price␣cannot␣be␣lower!")
```

Hint: What happens if one of the fields is `False`?

Example 3 — Invalid Domain Filter

```
self.env['estate.property'].search([('tag_ids.name', '=', 'Luxury')])
```

Hint: Can you filter on `many2many.name` like that?

Example 4 — Writing to a One2many Field Incorrectly

```
self.offer_ids = [{'partner_id': self.partner_id.id, 'price': 100000}]
```

Hint: One2many fields can't be assigned like this. Use commands.

Example 5 — Missing `return` in `create()`

```
@api.model
def create(self, vals):
    rec = super().create(vals)
    rec.reference = f"PROP-{rec.id}"
```

Hint: Where is the return?

Example 6 — Wrong Signature in Compute Method

```
@api.depends('offer_ids')
def _compute_offer_count(self, record):
    record.offer_count = len(record.offer_ids)
```

Hint: Compute methods always receive `self`.

Example 7 — Misusing `env.user`

```
for rec in self:
    rec.salesperson_id = self.env.user
```

Hint: Are you assigning a user or a recordset?

Example 8 — Wrong Use of `@api.depends`

```
@api.depends('offer_ids')
def _compute_offer_count(self):
    for rec in self:
        for offer in rec.offer_ids:
            offer.status = 'checked'
```

Hint: Compute methods should not modify other records.

Example 9 — Validation in `@api.onchange`

```
@api.onchange('expected_price')
def _onchange_price(self):
    if self.expected_price < 10000:
        raise ValidationError("Too␣low!")
```

Hint: Is `ValidationError` valid here?

Example 10 — Infinite Recursion

```
@api.onchange('expected_price')
def _onchange_expected_price(self):
    self.expected_price = self.expected_price + 100
```

Hint: What happens every time the field changes?

Part 4 — Build a Training Marketplace

Let's take our model further and simulate a real training marketplace. Each course now has levels, prices, tags, and ratings — and participants have credits and preferences.

Model: `training.course`

Basic Fields:

- `name` (Char)
- `description` (Text)
- `start_date`, `end_date` (Date)
- `trainer_id` (Many2one on res.partner)
- `price_credits` (Integer): cost in virtual credits
- `level` (Selection): `beginner`, `intermediate`, `expert`
- `tags` (Many2many): e.g., "Python", "Odoo", "Finance", "Soft Skills" (attention, this will be another model that doesn't exist yet !)
- `rating` (Float): average feedback score (0.0–5.0)
- `max_attendees` (Integer)
- `active` (Boolean)

Computed Fields:

- `duration_days` (integer)
- `available_slots` (integer)

Business Rules:

- If the course rating is under 2.5, show a warning in the form view. Showing a warning can be a bit painful, you can also just make a compute that will add a note if there is no note yet.
- Ensure `max_attendees > 0` and `price_credits >= 0`
- Mark courses with `rating > 4.5` and `price_credits < 50` as "hot deals" (Boolean field)

Part 5 — Logic & Optimization Challenges

Use the Odoo Shell or computed methods in your model to solve these logic problems using Python.

- In the shell, list the top 3 expert-level courses under 50 credits with at least 2 slots available.
- In the shell, showcase a dictionary `{tag: average_rating}` for all tags in use, for reporting purpose
- Write a function to compute a course score using this formula:

  `score = (rating * 10) + available_slots - price_credits`

  Sort all courses by this score and print the top 5.
- For each trainer, print how many students they have across all their courses. You can add it to their contact profile
- Suggest a course bundle to a user based on their preferred tags (simulate their preferences).

Part 6 — Enrollments and Feedback

Add a new model to handle course enrollments and simulate feedback mechanisms.

Model: `training.participant`

This model links a partner to a course they are attending.

Fields:

- `partner_id` (Many2one)
- `course_id` (Many2one)
- `enrolled_on` (Datetime)
- `feedback` (Text)

- `feedback`$_{s}core(Integer)$

Logic ideas:

- Use feedback scores to update the `rating` on `training.course`
- Add a button to simulate "Give Feedback"
- Prevent duplicate enrollments using a constraint

Bonus: Add an action to mass assign partners to courses if available slots exist.