<https://www.youtube.com/watch?v=8V-uOG8KkKA&list=PLqRRLx0cl0hphrDY9cvs0ZEUn-VZd9dcP>

**all method are inside a models.py**

D:\Projects\v18\odoo\odoo\models.py

**1. search Method**

The search method is used to find records that match specific criteria in a model. It is commonly used in Odoo's ORM (Object-Relational Mapping) to query data efficiently.

**Syntax:**

records = self.env['model.name'].search(domain, limit=0, order=None)

or

records = self.search(domain, limit=0, order=None) # if method define in current model

**Parameters:**

1. **domain** (list) – Specifies the conditions for filtering records.
2. **limit** (int, optional) – Limits the number of results returned. Default is 0 (no limit).
3. **order** (str, optional) – Orders the results based on a field.

**Returns:**

A recordset containing matching records.

**Examples:**

**1. Fetch all records from a model**

def get\_customers(self):

customers = self.env['res.partner'].search([])

print(customers)

**2. Search with a domain filter**

customers = self.env['res.partner'].search([('customer\_rank', '>', 0)])

**3. Search with multiple conditions**

customers = self.env['res.partner'].search([

('customer\_rank', '>', 0),

('city', '=', 'New York')

])

**4. Search with a limit**

customers = self.env['res.partner'].search([], limit=5)

**5. Search with ordering**

customers = self.env['res.partner'].search([], order='name asc')

**Difference Between search and search\_read**

* search() returns a recordset.
* search\_read() fetches the records along with specified fields.

**Example:**

customers = self.env['res.partner'].search\_read([], ['name', 'email'])

**2. search\_count Method**

The search\_count method in Odoo is used to count the number of records that match a given domain filter. It is more efficient than search because it does not retrieve actual records—only the count.

**Syntax:**

count = self.env['model.name'].search\_count(domain)

or

count = self.search\_count(domain) # if method define in current model

**Parameters:**

**1. domain (list) –** Specifies the conditions for filtering records.

**Returns:**

An integer representing the number of matching records.

**Examples:**

**1. Count all records in a model**

def get\_customer(self):

customer\_count = self.env['res.partner'].search\_count([])

print(customer\_count)

**2. Count records with a condition**

customer\_count = self.env['res.partner'].search\_count([('customer\_rank', '>', 0)])

print(customer\_count)

**3. Count records with multiple conditions**

customer\_count = self.env['res.partner'].search\_count([

('customer\_rank', '>', 0),

('city', '=', 'New York')

])

print(customer\_count)

**Difference Between search and search\_count**

| **Method** | **Purpose** | **Returns** |
| --- | --- | --- |
| search() | Retrieves recordsets that match the domain | Recordset |
| search\_count() | Counts matching records without fetching them | Integer |

**Example:**

customers = self.env['res.partner'].search([('customer\_rank', '>', 0)])

print(len(customers)) **# Same as search\_count but less efficient**

customer\_count = self.env['res.partner'].search\_count([('customer\_rank', '>', 0)])

print(customer\_count) **# More efficient**

**3. ref Method**

Use ref method to first understand about over module **data records** .

**data/**

└── data.xml # This is where you define records (not inside views)

**Manifest File Configuration:**

'data': [ 'data/data.xml', # Load records from the data folder ],

**Example of data record:**

<odoo>

<record **id="example\_partner"** model="model.name">

<field name="name">Example Partner</field>

<field name="email">example@partner.com</field>

<field name="phone">+1234567890</field>

<field name="is\_company">True</field>

<field name="company\_type">company</field>

</record>

</odoo>

The **ref** method in Odoo is used to retrieve a record by its external XML ID (record reference). It is commonly used to get records defined in XML files, such as default configurations, user groups, or specific records.

**Syntax:**

record = self.env.ref(**'module\_name**.xml\_id', raise\_if\_not\_found=True)

Note: use module name not a model name

**Parameters:**

* **module\_name.xml\_id** (str) – The full external identifier of the record.
* **raise\_if\_not\_found** (bool, optional) – If True, an error is raised if the record is not found; otherwise, None is returned. Default is True.

**Returns:**

A **recordset** containing the referenced record.

**Examples:**

**1.** def get\_default\_partner(self):

partner = self.env.ref(module\_name.xml\_id')

print(partner.name)

return partner.name

**2. Get a Specific Record by XML ID**

currency = self.env.ref('base.USD')

print(currency.name) # Output: US Dollar

**3. Get a User Group**

group\_manager = self.env.ref('base.group\_system')

print(group\_manager.name) # Output: Settings

**4. Handle Missing Records**

record = self.env.ref('my\_module.some\_record', raise\_if\_not\_found=False)

if record:

print("Record found:", record.name)

else:

print("Record not found")

**When to Use ref?**

* Fetching default system records (e.g., currencies, countries, user groups).
* Accessing records created via XML data files.
* Avoiding hardcoded database IDs.

**4. browse Method**

the browse method is used to retrieve specific records from the database by their unique IDs. It allows you to access existing records directly without applying any search filters. This method is often used when you already know the IDs of the records you need and want to avoid searching through all the records.

**Syntax:**

record = self.env['model.name'].browse(record\_ids)

or

record = self.browse(record\_ids) # if method define in current model

**Parameters:**

* **model.name**: The model from which you want to retrieve records (e.g., res.partner, product.product).
* **record\_ids**: A list of record IDs or a single record ID. These IDs are the primary keys of the records in the model.

**Returns:**

The method returns a **recordset**. Even if you provide a single ID, it will still return a recordset (which can be empty if the record doesn't exist).

**Examples:**

**1. Browse a Single Record**

def custom\_method(self):

partner = self.env['res.partner'].browse(1)

print(partner.name) # Access the 'name' field of the partner with ID 1

**2. Browse Multiple Records**

partners = self.env['res.partner'].browse([1, 2, 3])

for partner in partners:

print(partner.name)

**3. Handle Non-Existent Records**

non\_existent\_partner = self.env['res.partner'].browse(999999)

print(non\_existent\_partner) # This will print an empty recordset, not an error

partner = self.env['res.partner'].browse(1)

if partner.exists():

print("Partner exists:", partner.name)

else:

print("Partner does not exist.")

**Why Use browse?**

* **Efficient Access by ID**: When you already know the record's ID(s), using browse is more efficient than search because it directly accesses the record.

**5. exists Method**

The exists() method in Odoo is used to check whether a record (or multiple records) retrieved using browse() or search() actually exists in the database. This helps prevent errors when working with records that may have been deleted.

**Syntax:**

recordset.exists()

**Returns:** A recordset containing only the existing records. If all records are deleted or don't exist, it returns an emptyrecordset instead of raising an error.

**Example:**

**1. Checking if a Single Record Exists**

partner = self.env['res.partner'].browse(10) # Try to get partner with ID 10

if partner.exists():

print("Partner exists:", partner.name)

else:

print("Partner does not exist.")

**2. Checking if Multiple Records Exist**

partners = self.env['res.partner'].browse([5, 10, 15]) # Fetch multiple IDs

existing\_partners = partners.exists()

print("Existing Partners:", existing\_partners)

**3. Safe to update**

partner = self.env['res.partner'].browse(1000) # ID 1000 may not exist

if partner.exists():

partner.write({'email': 'test@example.com'}) # Safe to update

else:

print("Cannot update, partner does not exist!")

**4. Filtered Search**

partners = self.env['res.partner'].search([('email', '=', 'test@example.com')])

existing\_partners = partners.exists()

print(existing\_partners) # Prints only the partners that still exist

**Why Use exists()?**

* Prevents errors when trying to access a non-existent record.
* Avoids unnecessary database queries.
* Ensures the record you are working with is still in the database.

**6. create Method**

The create() method in Odoo is used to **insert (create) new records** in a model. It takes a dictionary (or a list of dictionaries) with field values and returns the newly created record(s).

**Syntax:**

new\_record = self.env['model.name'].create({

'field\_name1': value1,

'field\_name2': value2,

})

**Or**

new\_record = self.create({

'field\_name1': value1,

'field\_name2': value2,

}) # if method define in current model

**If you override the create method in your current model**

@api.model

def create(self, vals):

# Custom logic before record creation

record = super(ModelName, self).create(vals)

# Custom logic after record creation (optional)

return record

**Returns**: The newly created record (as a recordset).

**Example:**

**1. Creating a Single Record**

new\_partner = self.env['res.partner'].create({

'name': 'Rahul Dudhrejiya',

'email': 'rahul@example.com',

'phone': '+1234567890',

'company\_type': 'person',

})

print(new\_partner.name) # Output: Rahul Dudhrejiya

**2. Creating Multiple Records**

partners = self.env['res.partner'].create([

{'name': 'Alice', 'email': 'alice@example.com'},

{'name': 'Bob', 'email': 'bob@example.com'},

])

print(partners) # This will contain both records

**Override create Method:**

override create method when you click a save button.

**1. Adding Custom Logic Before Creation**

from odoo import models, fields, api

class **CustomModel**(models.Model):

\_name = 'custom.model'

\_description = 'Custom Model'

name = fields.Char(string="Name", required=True)

description = fields.Text(string="Description")

@api.model

def create(self, **vals**):

# Add default description if not provided

if 'description' not in vals or not vals['description']:

vals['description'] = "Default Description"

# Call the original create method

record = super(**CustomModel**, self).create(**vals**)

# Additional actions after record creation

print(f"New Record Created: {record.name}")

return record

**2. Generating a Unique Reference Number**

from odoo import models, fields, api

class CustomOrder(models.Model):

\_name = 'custom.order'

\_description = 'Custom Order'

name = fields.Char(string="Customer Name", required=True)

ref = fields.Char(string="Reference", readonly=True)

@api.model

def create(self, vals):

# Generate a unique reference number

vals['ref'] = self.env['ir.sequence'].next\_by\_code('custom.order') or 'New'

# Call the super method to create the record

record = super(CustomOrder, self).create(vals)

return record

**3. Prevent Duplicate Records**

from odoo import models, fields, api, exceptions

class UniqueCustomer(models.Model):

\_name = 'unique.customer'

\_description = 'Unique Customer'

name = fields.Char(string="Customer Name", required=True)

@api.model

def create(self, vals):

# Check if a record with the same name already exists

existing = self.env['unique.customer'].search([('name', '=', vals.get('name'))])

if existing:

raise exceptions.ValidationError(f"A customer with the name {vals.get('name')} already exists!")

return super(UniqueCustomer, self).create(vals)

**📌 Summary**

| **Method** | **Input Type** | **Use Case** |
| --- | --- | --- |
| @api.model | vals (dictionary) | Creating a single record |
| @api.model\_create\_multi | vals\_list (list of dictionaries) | Creating multiple records at once |

**7. write Method**

The write() method is used to update existing records in the database, and it also allows you to add custom logic when updating records.

**Syntax:**

record.write({

'field\_name1': value1,

'field\_name2': value2,

})

**If you override the create method in your current model**

@api.model

def write(self, vals):

# Custom logic before updating the record

return super(ModelName, self).write(vals)

**Returns**:

The write method returns a **boolean**:

* **True**: Indicates the update was successful.
* **False**: Indicates the update failed.

**Example:**

**1. Basic write usage**

# Retrieve a record by its ID

partner = self.env['res.partner'].browse(1) # Fetch partner record with ID 1

# Update the 'name' and 'email' fields of the partner record

partner.write({

'name': 'New Name',

'email': 'new\_email@example.com',

})

print(partner.name) # Output: New Name

**2. Updating Multiple Records**

# Search for all partners where the company\_type is 'person'

partners = self.env['res.partner'].search([('company\_type', '=', 'person')])

# Update the 'phone' field for all found partners

partners.write({

'phone': '+9876543210'

})

**Override write Method:**

You can override the write() method in your own model to add custom logic before or after updating a record

**1. Custom validation before updating a record**

from odoo import models, fields, api, exceptions

class CustomPartner(models.Model):

\_name = 'custom.partner'

\_description = 'Custom Partner'

name = fields.Char(string="Name")

email = fields.Char(string="Email")

phone = fields.Char(string="Phone")

def write(self, vals):

# Add custom validation logic before writing data

if 'phone' in vals and len(vals['phone']) != 10:

raise exceptions.ValidationError("Phone number must be 10 digits long.")

# Call the original `write` method to update the record

return super(CustomPartner, self).write(vals)

**2. Logging and Additional Actions After Writing**

from odoo import models, fields, api

class CustomPartner(models.Model):

\_name = 'custom.partner'

\_description = 'Custom Partner'

name = fields.Char(string="Name")

email = fields.Char(string="Email")

phone = fields.Char(string="Phone")

def write(self, vals):

# Call the original `write` method to update the record

result = super(CustomPartner, self).write(vals)

# Custom logic after writing (e.g., log the update)

if result:

print(f"Record {self.id} updated successfully")

return result

**8. copy method**

the copy() method is used to duplicate an existing record. By default, when you duplicate a record (e.g., via the "Duplicate" button in the UI or programmatically), Odoo calls the copy() method of the model. You can override this method to customize the duplication behavior.

**Syntax:**

new\_record = record.copy(default=None)

**default Parameter**:

The default argument is a **dictionary** of field-value pairs to override during duplication.

**Note:** Fields marked with copy=False in their definition are not copied.

**Example:**

# Duplicate a record

original\_record = self.env['my.model'].browse(1)

new\_record = original\_record.copy()

# Duplicate with overrides

new\_record = original\_record.copy({'name': 'Custom Name'})

**When to Override copy()**

* To customize field values during duplication.
* To add validation before copying.
* To skip copying specific related records.

**Override the copy() Method**

from odoo import models, fields, api

class MyModel(models.Model):

\_name = 'my.model'

\_description = 'My Model'

name = fields.Char(string='Name', required=True, copy=True)

reference = fields.Char(string='Reference', copy=False) # Not copied by default

active = fields.Boolean(string='Active', default=True)

@api.returns('self', lambda value: value.id)

def copy(self, default=None):

# Add custom logic here

default = dict(default or {}) # Initialize default as a dictionary

# Example: Append "(Copy)" to the name

default['name'] = self.name + " (Copy)"

# Example: Reset the 'active' field to True

default['active'] = True

# Call the original copy method with the updated 'default' values

return super(MyModel, self).copy(default)

**Custom Copy Logic**

class SaleOrder(models.Model):

\_inherit = 'sale.order'

def copy(self, default=None):

default = dict(default or {})

# Append "(Copy)" to the order name

default['name'] = self.name + " (Copy)"

# Reset the state to 'draft'

default['state'] = 'draft'

# Do NOT copy the client\_order\_ref field

default['client\_order\_ref'] = False

return super(SaleOrder, self).copy(default)

**9. unlink Method**

The unlink() method in Odoo is used to delete one or more records from the database. This method removes the records permanently and is typically used to handle the deletion of records in models.

**Syntax:**

record.unlink()

**Returns:** The unlink() method returns True if the records were successfully deleted. If the deletion fails (due to constraints or permissions), it will raise an exception.

**Example :**

**1. Basic unlink() Usage**

# Retrieve a record by its ID

partner = self.env['res.partner'].browse(1) # Fetch partner record with ID 1

# Delete the partner record

partner.unlink()

# After this, partner with ID 1 will no longer exist in the database.

**2. Deleting Multiple Records**

# Search for all partners with a specific condition

partners = self.env['res.partner'].search([('company\_type', '=', 'person')])

# Delete all the found partners

partners.unlink()

# All partners with company\_type 'person' will be deleted.

**Overriding the unlink Method:**

You can override the unlink() method to add custom logic before or after deleting records, such as adding validation or logging actions.

**1. Custom Validation Before Deleting a Record**

from odoo import models, fields, api, exceptions

class CustomPartner(models.Model):

\_name = 'custom.partner'

\_description = 'Custom Partner'

name = fields.Char(string="Name")

email = fields.Char(string="Email")

phone = fields.Char(string="Phone")

def unlink(self):

# Custom validation before deleting a record

if self.name == 'Critical Partner':

raise exceptions.UserError("You cannot delete a critical partner record.")

# Call the original `unlink` method to delete the record

return super(CustomPartner, self).unlink()

**2. Logging or Additional Actions After Deleting**

from odoo import models, fields, api

class CustomPartner(models.Model):

\_name = 'custom.partner'

\_description = 'Custom Partner'

name = fields.Char(string="Name")

email = fields.Char(string="Email")

phone = fields.Char(string="Phone")

def unlink(self):

# Custom logic before deleting (e.g., logging the deletion)

for record in self:

print(f"Record {record.id} with name {record.name} will be deleted.")

# Call the original `unlink` method to delete the record

return super(CustomPartner, self).unlink()

**10. mapped Method**

The mapped() method in Odoo is used to retrieve the values of a specific field across a recordset and return them as a list. This method allows you to extract a particular field from multiple records at once, **avoiding the need for a loop**.

**Syntax:**

recordset.mapped(field\_name)

* recordset: The recordset of model instances on which you want to operate.
* field\_name: The name of the field (or related field) you want to extract values for.

The mapped() method will return a list of values for the specified field. If the field is a relational field (like a Many2one, One2many, Many2many), the result may be a list of related recordsets.

**Example:**

**1. Basic Usage of mapped()**

# Retrieve a list of partner names

partners = self.env['res.partner'].search([]) # Get all partner records

names = partners.mapped('name') # Extract the 'name' field from all partner records

print(names)

Output: ['John Doe', 'Jane Smith', 'Mike Johnson']

**2. Using mapped() with Related Fields**

If you're working with relational fields (e.g., Many2one, One2many, Many2many), you can use mapped() to access related fields.

# Retrieve a list of email addresses of the contacts associated with the partners

partners = self.env['res.partner'].search([])

# Get all partner records

emails = partners.mapped('child\_ids.email')

# Extract the 'email' field from the related 'child\_ids' field (Many2one relationship)

print(emails)

Output: ['child1@example.com', 'child2@example.com']

**3. Using mapped() with Multiple Fields**

# Retrieve a list of tuples containing the partner name and email

partners = self.env['res.partner'].search([]) # Get all partner records

partner\_info = partners.mapped(lambda p: (p.name, p.email)) # Using lambda to return both name and email

print(partner\_info)

Output: [('John Doe', 'john@example.com'), ('Jane Smith', 'jane@example.com')]

**11. sorted Method**

The sorted() method in Odoo is used to sort a recordset based on one or more fields. It allows you to order the records in ascending or descending order, making it easier to retrieve records in a specific order according to your needs.

**Syntax:**

recordset.sorted(key=None, reverse=False)

* key: The field or function to sort by. It can be the name of a field or a **lambda** **function**.
* reverse: A boolean value. If True, the recordset will be sorted in descending order; if False, it will be sorted in ascending order (default).

**Example:**

**1. Basic Usage of sorted()**

# Retrieve all partner records and sort them by 'name' in ascending order

partners = self.env['res.partner'].search([]) # Get all partner records

sorted\_partners = partners.sorted(key='name') # Sort by 'name' field in ascending order

Output : ['Alice', 'Bob', 'Charlie']

**2. Sorting in Descending Order**

# Sort the partners by 'name' in descending order

partners = self.env['res.partner'].search([]) # Get all partner records

sorted\_partners\_desc = partners.sorted(key='name', **reverse=True**) # Sort by 'name' field in descending order

Output: ['Charlie', 'Bob', 'Alice']

**3. Sorting by Multiple Fields**

# Sort by 'company\_type' first, and then by 'name'

partners = self.env['res.partner'].search([]) # Get all partner records

sorted\_partners\_multiple = partners.sorted(key=['company\_type', 'name'])

# the records will first be sorted by company\_type, and then by name within each company type.

**4. Sorting Using a Lambda Function**

# Sort by the length of the partner's name (ascending order)

partners = self.env['res.partner'].search([]) # Get all partner records

sorted\_partners\_by\_length = partners.sorted(key=lambda p: len(p.name))

**5. Sorting with Date Fields**

# Sort the partners by their creation date (oldest to newest)

partners = self.env['res.partner'].search([]) # Get all partner records

sorted\_partners\_by\_date = partners.sorted(key='create\_date')

# such as create\_date or any custom date field.

**Important Notes:**

* sorted() does not modify the original recordset. It returns a new sorted recordset.
* The sorting is done **in-memory** after the recordset is retrieved from the database, so be mindful of the size of the recordset when sorting large datasets.
* The key argument can also be used with related fields, allowing you to sort based on fields from related models.

**12. filtered Method**

The filtered method in Odoo is used to filter a recordset based on a condition. It works similarly to Python's filter() function and returns a **subset** of the original recordset that meets the specified condition.

**Syntax:**

filtered\_records = recordset.filtered(lambda rec: condition)

* **recordset**: A set of records (e.g., self.env['model.name'].search([]))
* **lambda** rec: condition: A lambda function that defines the condition to filter records
* **filtered\_records**: The new recordset containing only the records that satisfy the condition

**Example:**

**1. Filter Students Older Than 18**

students = self.env['practice.student'].search([]) # Get all student records

adult\_students = students.filtered(lambda student: student.age > 18)

**2. Filter Employees in a Specific Department**

employees = self.env['hr.employee'].search([])

it\_employees = employees.filtered(lambda emp: emp.department\_id.name == "IT")

**3. Filter Active Records**

active\_partners = self.env['res.partner'].search([]).filtered(lambda p: p.active)

**Difference Between filtered() and search()**

| **Feature** | **search()** | **filtered()** |
| --- | --- | --- |
| Executes SQL Query? | Yes | No |
| Works on All Records in DB? | Yes | No (Only on the given recordset) |
| Faster for Large Data? | Yes | No (May be slower) |

🔹 **Use search() for better performance** when filtering records from the database.  
🔹 **Use filtered() when working with an existing recordset** in memory.

**13. name\_get Method (Similar \_rec\_name attribute)**

The name\_get method in Odoo is used to define how a record's **display name** appears in selection fields, Many2one fields, and record representations in views.

**Default Behavior of name\_get**

By default, Odoo uses the name field as the display name in Many2one fields. However, you can override name\_get to customize how records are displayed.

**Syntax:**

def name\_get(self):

result = []

for record in self:

result.append((record.id, record.name)) # Default behavior

return result

* Returns a **list of tuples**, where each tuple contains (record\_id, display\_name).
* Can be customized to include other fields.

**Example:**

**1. Custom name\_get in a Student Model**

class Student(models.Model):

\_name = "practice.student"

name = fields.Char(string="Student Name", required=True)

age = fields.Integer(string="Age")

def name\_get(self):

result = []

for record in self:

name\_display = f"{record.name} ({record.age} years)"

result.append((record.id, name\_display))

return result

Output: John Doe (20 years)

Alice Smith (22 years)

**Where is name\_get Used?**

* Many2one fields (student\_id, employee\_id, etc.)
* Selection fields
* Smart buttons
* Odoo views (Kanban, Form, etc.)

**14. name\_serach Method (Similar to \_rec\_name\_search Attribute)**

The **name\_search** method in Odoo is used to **customize search behavior** for Many2one fields. By default, Odoo searches in the name field (or \_rec\_name if defined). However, by overriding name\_search, you can **extend** the search to include other fields or apply special filters.

**Default Behavior of name\_search**

* When you type in a **Many2one field**, Odoo searches for records based on \_rec\_name (default: name field).
* If \_rec\_name is **not** set, Odoo throws an error.
* name\_search allows you to modify this behavior.

**Syntax:**

def name\_search(self, name='', args=None, operator='ilike', limit=100):

# Custom search logic

return self.search(args, limit=limit).name\_get()

**Parameters:**

* **name** → User input in the Many2one search box.
* **args** → Additional domain filters.
* **operator** → Search operator (default: ilike, case-insensitive match).
* **limit** → Max records returned (default: 100).
* **Returns** → A list of tuples [(id, display\_name)], formatted using name\_get().

**Example:**

**1. Search by Both Name and Email**

class Student(models.Model):

\_name = "practice.student"

\_rec\_name = "name" # Default display name

name = fields.Char(string="Student Name", required=True)

email = fields.Char(string="Email")

def name\_search(self, name='', args=None, operator='ilike', limit=100):

args = args or []

if name:

args += ['|', ('name', operator, name), ('email', operator, name)]

return self.search(args, limit=limit).name\_get()

**How It Works:**

* If a user types "john", it will search **both** name and email.
* The operator='ilike' makes it **case-insensitive**.
* The **name\_get()** method is used to format the output.

**2. Search by Name + Filter Active Students Only**

class Student(models.Model):

\_name = "practice.student"

name = fields.Char(string="Student Name", required=True)

email = fields.Char(string="Email")

active = fields.Boolean(default=True) # Active students only

def name\_search(self, name='', args=None, operator='ilike', limit=100):

args = args or []

if name:

args += ['|', ('name', operator, name), ('email', operator, name)]

args.append(('active', '=', True)) # Filter only active students

return self.search(args, limit=limit).name\_get()

**3. Custom Display Format Using name\_get()**

class Student(models.Model):

\_name = "practice.student"

name = fields.Char(string="Student Name", required=True)

email = fields.Char(string="Email")

def name\_get(self):

result = []

for record in self:

display\_name = f"{record.name} (Email: {record.email})"

result.append((record.id, display\_name))

return result

Output: When searching for students, the dropdown will display

John Doe (Email: john@example.com)

Alice Smith (Email: [alice@example.com](mailto:alice@example.com))

**When Should You Use name\_search?**

✅ Use name\_search when:

* You need to search in **multiple fields** (e.g., name + email).
* You want to **filter** records dynamically (e.g., only active users).
* You want to **modify the search behavior** of Many2one fields.

**15. default\_get Method**

the default get method is used to retrieve the default values for fields when a new record is being created. It is defined in the models.Model class as default\_get.

**Syntax:**

@api.model

def default\_get(self, fields\_list):

pass

**How It Works:**

* It is automatically called when a new record is created (e.g., in the UI or via code).
* It retrieves default values for the fields specified in fields\_list.
* It first checks:
  1. Explicit defaults set by default\_ prefixes.
  2. Defaults defined in Python code using @api.model and default\_get.
  3. Defaults from context (e.g., default\_field\_name=value).
  4. Defaults from ir.default (system-wide defaults in Odoo).

**Example:**

**1. Override default\_get**

from odoo import models, fields, api

class CustomModel(models.Model):

\_name = 'custom.model'

\_description = 'Custom Model'

name = fields.Char(string="Name")

status = fields.Selection([('draft', 'Draft'), ('done', 'Done')], default='draft')

@api.model

def default\_get(self, fields\_list):

defaults = super().default\_get(fields\_list)

defaults['name'] = 'Default Name'

return defaults

**2. Setting Defaults via Context**

return {

'type': 'ir.actions.act\_window',

'name': 'Create Record',

'res\_model': 'custom.model',

'view\_mode': 'form',

'target': 'new',

'context': {'default\_status': 'done'}

}

**3. Setting Defaults Using default\_ Prefix**

status = fields.Selection([('draft', 'Draft'), ('done', 'Done')], default='draft')

**Or, dynamically when opening a form:**

self.env['custom.model'].create({'default\_status': 'done'})

**16. sudo Method**

the sudo() method is used to **bypass access rights** and **execute operations as the Superuser (Administrator)**. This is useful when you need to perform actions that a regular user might not have permission for.

**Syntax:**

recordset.sudo()

**Example 1:**

**Without sudo() (Permission Error)**

record = self.env['res.partner'].browse(1) # Fetch a record

record.write({'name': 'New Name'}) # Might fail if the user lacks permissions

* If the user **does not have write access**, Odoo will raise an **AccessError**.

**With sudo() (Bypass Access Rights)**

record = self.env['res.partner'].browse(1).sudo()

record.write({'name': 'New Name'}) # Works even if the user lacks permissions

* Now the operation runs **as the Superuser**, ignoring access rules.

**Example 2:**

user = self.env['res.users'].browse(5) # User with ID 5

record = self.env['res.partner'].browse(1).sudo(user.id)

* This runs the operation with **User ID 5’s** permissions.

**Example 3: Creating a Record with sudo()**

class CustomModel(models.Model):

\_name = 'custom.model'

name = fields.Char(string="Name")

def create\_record(self):

self.env['custom.model'].sudo().create({'name': 'Admin Created'})

**Where to Use sudo()**

✅ **Use Cases:**

* When performing system-wide updates.
* When a model needs modification but users lack permissions.
* When running scheduled jobs (cron tasks).

🚫 **Avoid Using sudo():**

* In public-facing APIs (may expose sensitive data).
* When enforcing security rules (overuse can create vulnerabilities).

**Key Takeaways**

1. **sudo() bypasses access rights** and runs operations as the Superuser.
2. **It should be used carefully** to avoid security risks.
3. **You can use sudo(user\_id)** to run as a specific user.
4. **Use it in backend operations** but avoid exposing it in APIs.

**17. name\_create Method**

the **name\_create** method is used to customize the behavior when creating a record from a string value, typically in fields like **Many2one** or **Selection** that allow users to create a new record directly from the user interface.

The **name\_create** method is implemented on the model and provides the ability to define how new records are created when the user enters a new value that doesn't already exist in the database.

**Syntax:**

@api.model

def name\_create(self, name):

# Custom logic to create a new record

record = self.create({

'name': name,

# any other necessary fields

})

return record.name\_get()[0]

* **name\_create method**: This method is invoked when a new record is created by a user input. It receives a single parameter, name, which is the string the user entered.
* **Creating a record**: Inside name\_create, you can define logic to create a new record using the create method.
* **name\_get**: After the record is created, you should return the display name of the record (usually using the name\_get method, which returns a tuple of ID and name).

**Example 1:**

Let's say you have a ProductCategory model and want to customize the creation of categories by name. You might write:

from odoo import models, fields

class ProductCategory(models.Model):

\_name = 'product.category'

coustom\_name = fields.Char('Category Name')

@api.model

def name\_create(self, name):

# Create a new category record with the given name

new\_category = self.create({' coustom\_name ': name})

return new\_category.name\_get()[0]

**Example 2:**

from odoo import models, fields

class ProductCategory(models.Model):

\_name = 'product.category'

name = fields.Char('Category Name')

@api.model

def name\_create(self, name):

# Modify the name before creating the record

modified\_name = name.upper() # Example: Convert the name to uppercase

# Create a new category record with the modified name

new\_category = self.create({'name': modified\_name})

# Return the display name using name\_get

return new\_category.name\_get()[0]

In this example:

* When the user types a new category name into a Many2one field (which links to product.category), the name\_create method is triggered.
* It creates a new category with the name provided.
* The name\_get method returns the name that will be displayed in the field.