```python
# sale_order.py
# Part of Odoo. See LICENSE file for full copyright and licensing details.
from collections import defaultdict
from datetime import timedelta
from itertools import groupby

from odoo import api, fields, models, SUPERUSER_ID, _
from odoo.exceptions import AccessError, UserError, ValidationError
from odoo.fields import Command
from odoo.osv import expression
from odoo.tools import float_is_zero, format_amount, format_date, html_keep_url,
is_html_empty
from odoo.tools.sql import create_index

from odoo.addons.payment import utils as payment_utils

INVOICE_STATUS = [
    ('upselling', 'Upselling Opportunity'),
    ('invoiced', 'Fully Invoiced'),
    ('to invoice', 'To Invoice'),
    ('no', 'Nothing to Invoice')
]

SALE_ORDER_STATE = [
    ('draft', "Quotation"),
    ('sent', "Quotation Sent"),
    ('sale', "Sales Order"),
    ('cancel', "Cancelled"),
]


class SaleOrder(models.Model):
    _name = 'sale.order'
    _inherit = ['portal.mixin', 'product.catalog.mixin', 'mail.thread',
'mail.activity.mixin', 'utm.mixin']
    _description = "Sales Order"
    _order = 'date_order desc, id desc'
    _check_company_auto = True

    _sql_constraints = [
        ('date_order_conditional_required',
         "CHECK((state = 'sale' AND date_order IS NOT NULL) OR state != 'sale')",
         "A confirmed sales order requires a confirmation date."),
    ]

    @property
    def _rec_names_search(self):
        if self._context.get('sale_show_partner_name'):
            return ['name', 'partner_id.name']
        return ['name']

    #=== FIELDS ===#

    name = fields.Char(
        string="Order Reference",
        required=True, copy=False, readonly=False,
        index='trigram',
        default=lambda self: _('New'))

    company_id = fields.Many2one(
        comodel_name='res.company',
        required=True, index=True,
        default=lambda self: self.env.company)
    partner_id = fields.Many2one(
        comodel_name='res.partner',
        string="Customer",
        required=True, change_default=True, index=True,
        tracking=1,
        domain="[('company_id', 'in', (False, company_id))]")
    state = fields.Selection(
        selection=SALE_ORDER_STATE,
        string="Status",
            readonly=True, copy=False, index=True,
            tracking=3,
            default='draft')
    locked = fields.Boolean(default=False, copy=False, help="Locked orders cannot be modified.")

    client_order_ref = fields.Char(string="Customer Reference", copy=False)
    create_date = fields.Datetime(  # Override of default create_date field from ORM
        string="Creation Date", index=True, readonly=True)
    commitment_date = fields.Datetime(
        string="Delivery Date", copy=False,
        help="This is the delivery date promised to the customer. "
             "If set, the delivery order will be scheduled based on "
             "this date rather than product lead times.")
    date_order = fields.Datetime(
        string="Order Date",
        required=True, copy=False,
        help="Creation date of draft/sent orders,\nConfirmation date of confirmed orders.",
        default=fields.Datetime.now)
    origin = fields.Char(
        string="Source Document",
        help="Reference of the document that generated this sales order request")
    reference = fields.Char(
        string="Payment Ref.",
        help="The payment communication of this sale order.",
        copy=False)

    require_signature = fields.Boolean(
        string="Online signature",
        compute='_compute_require_signature',
        store=True, readonly=False, precompute=True,
        help="Request a online signature from the customer to confirm the order.")
    require_payment = fields.Boolean(
        string="Online payment",
        compute='_compute_require_payment',
        store=True, readonly=False, precompute=True,
        help="Request a online payment from the customer to confirm the order.")
    prepayment_percent = fields.Float(
        string="Prepayment percentage",
        compute='_compute_prepayment_percent',
        store=True, readonly=False, precompute=True,
        help="The percentage of the amount needed that must be paid by the customer to confirm the order.")

    signature = fields.Image(
        string="Signature",
        copy=False, attachment=True, max_width=1024, max_height=1024)
    signed_by = fields.Char(
        string="Signed By", copy=False)
    signed_on = fields.Datetime(
        string="Signed On", copy=False)

    validity_date = fields.Date(
        string="Expiration",
        compute='_compute_validity_date',
        store=True, readonly=False, copy=False, precompute=True)
    journal_id = fields.Many2one(
        'account.journal', string="Invoicing Journal",
        compute="_compute_journal_id", store=True, readonly=False, precompute=True,
        domain=[('type', '=', 'sale')], check_company=True,
        help="If set, the SO will invoice in this journal; "
             "otherwise the sales journal with the lowest sequence is used.")

    # Partner-based computes
    note = fields.Html(
        string="Terms and conditions",
        compute='_compute_note',
        store=True, readonly=False, precompute=True)

    partner_invoice_id = fields.Many2one(
        comodel_name='res.partner',
        string="Invoice Address",
        compute='_compute_partner_invoice_id',
        store=True, readonly=False, required=True, precompute=True,
        domain="['|', ('company_id', '=', False), ('company_id', '=', company_id)]")
    partner_shipping_id = fields.Many2one(
        comodel_name='res.partner',
        string="Delivery Address",
        compute='_compute_partner_shipping_id',
        store=True, readonly=False, required=True, precompute=True,
        domain="['|', ('company_id', '=', False), ('company_id', '=', company_id)]",)

    fiscal_position_id = fields.Many2one(
        comodel_name='account.fiscal.position',
        string="Fiscal Position",
        compute='_compute_fiscal_position_id',
        store=True, readonly=False, precompute=True, check_company=True,
        help="Fiscal positions are used to adapt taxes and accounts for particular customers or sales orders/invoices."
            "The default value comes from the customer.",
        domain="[('company_id', '=', company_id)]")
    payment_term_id = fields.Many2one(
        comodel_name='account.payment.term',
        string="Payment Terms",
        compute='_compute_payment_term_id',
        store=True, readonly=False, precompute=True, check_company=True,  # Unrequired company
        domain="['|', ('company_id', '=', False), ('company_id', '=', company_id)]")
    pricelist_id = fields.Many2one(
        comodel_name='product.pricelist',
        string="Pricelist",
        compute='_compute_pricelist_id',
        store=True, readonly=False, precompute=True, check_company=True,  # Unrequired company
        tracking=1,
        domain="['|', ('company_id', '=', False), ('company_id', '=', company_id)]",
        help="If you change the pricelist, only newly added lines will be affected.")
    currency_id = fields.Many2one(
        comodel_name='res.currency',
        compute='_compute_currency_id',
        store=True,
        precompute=True,
        ondelete='restrict'
    )
    currency_rate = fields.Float(
        string="Currency Rate",
        compute='_compute_currency_rate',
        digits=(12, 6),
        store=True, precompute=True)
    user_id = fields.Many2one(
        comodel_name='res.users',
        string="Salesperson",
        compute='_compute_user_id',
        store=True, readonly=False, precompute=True, index=True,
        tracking=2,
        domain=lambda self: "[('groups_id', '=', {}), ('share', '=', False), ('company_ids', '=', company_id)]".format(
            self.env.ref("sales_team.group_sale_salesman").id
        ))
    team_id = fields.Many2one(
        comodel_name='crm.team',
        string="Sales Team",
        compute='_compute_team_id',
        store=True, readonly=False, precompute=True, ondelete="set null",
        change_default=True, check_company=True,  # Unrequired company
        tracking=True,
        domain="['|', ('company_id', '=', False), ('company_id', '=', company_id)]")

    # Lines and line based computes
    order_line = fields.One2many(
        comodel_name='sale.order.line',
        inverse_name='order_id',
        string="Order Lines",
        copy=True, auto_join=True)

    amount_untaxed = fields.Monetary(string="Untaxed Amount", store=True, compute='_compute_amounts', tracking=5)
    amount_tax = fields.Monetary(string="Taxes", store=True, compute='_compute_amounts')
    amount_total = fields.Monetary(string="Total", store=True, compute='_compute_amounts', tracking=4)
    amount_to_invoice = fields.Monetary(string="Amount to invoice", store=True, compute='_compute_amount_to_invoice')
    amount_invoiced = fields.Monetary(string="Already invoiced", compute='_compute_amount_invoiced')

    invoice_count = fields.Integer(string="Invoice Count", compute='_get_invoiced')
    invoice_ids = fields.Many2many(
        comodel_name='account.move',
        string="Invoices",
        compute='_get_invoiced',
        search='_search_invoice_ids',
        copy=False)
    invoice_status = fields.Selection(
        selection=INVOICE_STATUS,
        string="Invoice Status",
        compute='_compute_invoice_status',
        store=True)

    # Payment fields
    transaction_ids = fields.Many2many(
        comodel_name='payment.transaction',
        relation='sale_order_transaction_rel', column1='sale_order_id', column2='transaction_id',
        string="Transactions",
        copy=False, readonly=True)
    authorized_transaction_ids = fields.Many2many(
        comodel_name='payment.transaction',
        string="Authorized Transactions",
        compute='_compute_authorized_transaction_ids',
        copy=False,
        compute_sudo=True)
    amount_paid = fields.Float(compute='_compute_amount_paid', compute_sudo=True)

    # UTMs - enforcing the fact that we want to 'set null' when relation is unlinked
    campaign_id = fields.Many2one(ondelete='set null')
    medium_id = fields.Many2one(ondelete='set null')
    source_id = fields.Many2one(ondelete='set null')

    # Followup ?
    analytic_account_id = fields.Many2one(
        comodel_name='account.analytic.account',
        string="Analytic Account",
        copy=False, check_company=True,  # Unrequired company
        domain="['|', ('company_id', '=', False), ('company_id', '=', company_id)]")
    tag_ids = fields.Many2many(
        comodel_name='crm.tag',
        relation='sale_order_tag_rel', column1='order_id', column2='tag_id',
        string="Tags")

    # Remaining non stored computed fields (hide/make fields readonly, ...)
    amount_undiscounted = fields.Float(
        string="Amount Before Discount",
        compute='_compute_amount_undiscounted', digits=0)
    country_code = fields.Char(related='company_id.account_fiscal_country_id.code', string="Country code")
    expected_date = fields.Datetime(
        string="Expected Date",
        compute='_compute_expected_date', store=False,  # Note: can not be stored since depends on today()
        help="Delivery date you can promise to the customer, computed from the minimum lead time of the order lines.")
    is_expired = fields.Boolean(string="Is Expired", compute='_compute_is_expired')
    partner_credit_warning = fields.Text(
        compute='_compute_partner_credit_warning')
    tax_calculation_rounding_method = fields.Selection(
        related='company_id.tax_calculation_rounding_method',
        depends=['company_id'])
    tax_country_id = fields.Many2one(
        comodel_name='res.country',
        compute='_compute_tax_country_id',
        # Avoid access error on fiscal position when reading a sale order with company != user.company_ids
        compute_sudo=True)  # used to filter available taxes depending on the fiscal country and position
    tax_totals = fields.Binary(compute='_compute_tax_totals', exportable=False)
    terms_type = fields.Selection(related='company_id.terms_type')
    type_name = fields.Char(string="Type Name", compute='_compute_type_name')

    # Remaining ux fields (not computed, not stored)

    show_update_fpos = fields.Boolean(
        string="Has Fiscal Position Changed", store=False)  # True if the fiscal position was changed
    has_active_pricelist = fields.Boolean(
        compute='_compute_has_active_pricelist')
    show_update_pricelist = fields.Boolean(
        string="Has Pricelist Changed", store=False)  # True if the pricelist was changed

    def init(self):
        create_index(self._cr, 'sale_order_date_order_id_idx', 'sale_order', ["date_order desc", "id desc"])

    #=== COMPUTE METHODS ===#

    @api.depends('partner_id')
    @api.depends_context('sale_show_partner_name')
    def _compute_display_name(self):
        if not self._context.get('sale_show_partner_name'):
            return super()._compute_display_name()
        for order in self:
            name = order.name
            if order.partner_id.name:
                name = f'{name} - {order.partner_id.name}'
            order.display_name = name

    @api.depends('company_id')
    def _compute_require_signature(self):
        for order in self:
            order.require_signature = order.company_id.portal_confirmation_sign

    @api.depends('company_id')
    def _compute_require_payment(self):
        for order in self:
            order.require_payment = order.company_id.portal_confirmation_pay

    @api.depends('require_payment')
    def _compute_prepayment_percent(self):
        for order in self:
            order.prepayment_percent = order.company_id.prepayment_percent

    @api.depends('company_id')
    def _compute_validity_date(self):
        today = fields.Date.context_today(self)
        for order in self:
            days = order.company_id.quotation_validity_days
            if days > 0:
                order.validity_date = today + timedelta(days)
            else:
                order.validity_date = False

    def _compute_journal_id(self):
        self.journal_id = False

    @api.depends('partner_id')
    def _compute_note(self):
        use_invoice_terms = self.env['ir.config_parameter'].sudo().get_param('account.use_invoice_terms')
        if not use_invoice_terms:
            return
        for order in self:
            order = order.with_company(order.company_id)
            if order.terms_type == 'html' and self.env.company.invoice_terms_html:
                baseurl = html_keep_url(order._get_note_url() + '/terms')
                context = {'lang': order.partner_id.lang or self.env.user.lang}
                order.note = _('Terms & Conditions: %s', baseurl)
                del context
            elif not is_html_empty(self.env.company.invoice_terms):
                order.note = order.with_context(lang=order.partner_id.lang).env.company.invoice_terms

    @api.model
    def _get_note_url(self):
        return self.env.company.get_base_url()

    @api.depends('partner_id')
    def _compute_partner_invoice_id(self):
        for order in self:
            order.partner_invoice_id = order.partner_id.address_get(['invoice'])['invoice'] if order.partner_id else False

    @api.depends('partner_id')
    def _compute_partner_shipping_id(self):
        for order in self:
            order.partner_shipping_id = order.partner_id.address_get(['delivery'])['delivery'] if order.partner_id else False

    @api.depends('partner_shipping_id', 'partner_id', 'company_id')
    def _compute_fiscal_position_id(self):
        """
        Trigger the change of fiscal position when the shipping address is modified.
        """
        cache = {}
        for order in self:
            if not order.partner_id:
                order.fiscal_position_id = False
                continue
            key = (order.company_id.id, order.partner_id.id, order.partner_shipping_id.id)
            if key not in cache:
                cache[key] = self.env['account.fiscal.position'].with_company(
                    order.company_id
                )._get_fiscal_position(order.partner_id, order.partner_shipping_id)
            order.fiscal_position_id = cache[key]

    @api.depends('partner_id')
    def _compute_payment_term_id(self):
        for order in self:
            order = order.with_company(order.company_id)
            order.payment_term_id = order.partner_id.property_payment_term_id

    @api.depends('partner_id', 'company_id')
    def _compute_pricelist_id(self):
        for order in self:
            if order.state != 'draft':
                continue
            if not order.partner_id:
                order.pricelist_id = False
                continue
            order = order.with_company(order.company_id)
            order.pricelist_id = order.partner_id.property_product_pricelist

    @api.depends('pricelist_id', 'company_id')
    def _compute_currency_id(self):
        for order in self:
            order.currency_id = order.pricelist_id.currency_id or order.company_id.currency_id

    @api.depends('currency_id', 'date_order', 'company_id')
    def _compute_currency_rate(self):
        for order in self:
            order.currency_rate = self.env['res.currency']._get_conversion_rate(
                from_currency=order.company_id.currency_id,
                to_currency=order.currency_id,
                company=order.company_id,
                date=order.date_order.date(),
            )

    @api.depends('company_id')
    def _compute_has_active_pricelist(self):
        for order in self:
            order.has_active_pricelist = bool(self.env['product.pricelist'].search(
                [('company_id', 'in', (False, order.company_id.id)), ('active', '=', True)],
                limit=1,
            ))

    @api.depends('partner_id')
    def _compute_user_id(self):
        for order in self:
            if order.partner_id and not (order._origin.id and order.user_id):
                # Recompute the salesman on partner change
                #   * if partner is set (is required anyway, so it will be set sooner or later)
                #   * if the order is not saved or has no salesman already
                order.user_id = (
                    order.partner_id.user_id
                    or order.partner_id.commercial_partner_id.user_id
                    or (self.user_has_groups('sales_team.group_sale_salesman') and self.env.user)
                )

    @api.depends('partner_id', 'user_id')
    def _compute_team_id(self):
        cached_teams = {}
        for order in self:
            default_team_id = self.env.context.get('default_team_id', False) or order.team_id.id or order.partner_id.team_id.id
            user_id = order.user_id.id
            company_id = order.company_id.id
            key = (default_team_id, user_id, company_id)
            if key not in cached_teams:
                cached_teams[key] = self.env['crm.team'].with_context(
                    default_team_id=default_team_id,
                )._get_default_team_id(
                    user_id=user_id,
                    domain=self.env['crm.team']._check_company_domain(company_id),
                )
            order.team_id = cached_teams[key]

    @api.depends('order_line.price_subtotal', 'order_line.price_tax', 'order_line.price_total')
    def _compute_amounts(self):
        """Compute the total amounts of the SO."""
        for order in self:
            order_lines = order.order_line.filtered(lambda x: not x.display_type)

            if order.company_id.tax_calculation_rounding_method == 'round_globally':
                tax_results = self.env['account.tax']._compute_taxes([
                    line._convert_to_tax_base_line_dict()
                    for line in order_lines
                ])
                totals = tax_results['totals']
                amount_untaxed = totals.get(order.currency_id, {}).get('amount_untaxed', 0.0)
                amount_tax = totals.get(order.currency_id, {}).get('amount_tax', 0.0)
            else:
                amount_untaxed = sum(order_lines.mapped('price_subtotal'))
                amount_tax = sum(order_lines.mapped('price_tax'))

            order.amount_untaxed = amount_untaxed
            order.amount_tax = amount_tax
            order.amount_total = order.amount_untaxed + order.amount_tax

    @api.depends('order_line.invoice_lines')
    def _get_invoiced(self):
        # The invoice_ids are obtained thanks to the invoice lines of the SO
        # lines, and we also search for possible refunds created directly from
        # existing invoices. This is necessary since such a refund is not
        # directly linked to the SO.
        for order in self:
            invoices = order.order_line.invoice_lines.move_id.filtered(lambda r: r.move_type in ('out_invoice', 'out_refund'))
            order.invoice_ids = invoices
            order.invoice_count = len(invoices)

    def _search_invoice_ids(self, operator, value):
        if operator == 'in' and value:
            self.env.cr.execute("""
                SELECT array_agg(so.id)
                    FROM sale_order so
                    JOIN sale_order_line sol ON sol.order_id = so.id
                    JOIN sale_order_line_invoice_rel soli_rel ON soli_rel.order_line_id = sol.id
                    JOIN account_move_line aml ON aml.id = soli_rel.invoice_line_id
                    JOIN account_move am ON am.id = aml.move_id
                WHERE
                    am.move_type in ('out_invoice', 'out_refund') AND
                    am.id = ANY(%s)
            """, (list(value),))
            so_ids = self.env.cr.fetchone()[0] or []
            return [('id', 'in', so_ids)]
        elif operator == '=' and not value:
            # special case for [('invoice_ids', '=', False)], i.e. "Invoices is not set"
            #
            # We cannot just search [('order_line.invoice_lines', '=', False)]
            # because it returns orders with uninvoiced lines, which is not
            # same "Invoices is not set" (some lines may have invoices and some
            # doesn't)
            #
            # A solution is making inverted search first ("orders with invoiced
            # lines") and then invert results ("get all other orders")
            #
            # Domain below returns subset of ('order_line.invoice_lines', '!=', False)
            order_ids = self._search([
                ('order_line.invoice_lines.move_id.move_type', 'in', ('out_invoice', 'out_refund'))
            ])
            return [('id', 'not in', order_ids)]
        return [
            ('order_line.invoice_lines.move_id.move_type', 'in', ('out_invoice', 'out_refund')),
            ('order_line.invoice_lines.move_id', operator, value),
        ]

    @api.depends('state', 'order_line.invoice_status')
    def _compute_invoice_status(self):
        """
        Compute the invoice status of a SO. Possible statuses:
        - no: if the SO is not in status 'sale' or 'done', we consider that there is nothing to
          invoice. This is also the default value if the conditions of no other status is met.
        - to invoice: if any SO line is 'to invoice', the whole SO is 'to invoice'
        - invoiced: if all SO lines are invoiced, the SO is invoiced.
        - upselling: if all SO lines are invoiced or upselling, the status is upselling.
        """
        confirmed_orders = self.filtered(lambda so: so.state == 'sale')
        (self - confirmed_orders).invoice_status = 'no'
        if not confirmed_orders:
            return
        line_invoice_status_all = [
            (order.id, invoice_status)
            for order, invoice_status in self.env['sale.order.line']._read_group([
                    ('order_id', 'in', confirmed_orders.ids),
                    ('is_downpayment', '=', False),
                    ('display_type', '=', False),
                ],
                ['order_id', 'invoice_status'])]
        for order in confirmed_orders:
            line_invoice_status = [d[1] for d in line_invoice_status_all if d[0] == order.id]
            if order.state != 'sale':
                order.invoice_status = 'no'
            elif any(invoice_status == 'to invoice' for invoice_status in line_invoice_status):
                order.invoice_status = 'to invoice'
            elif line_invoice_status and all(invoice_status == 'invoiced' for invoice_status in line_invoice_status):
                order.invoice_status = 'invoiced'
            elif line_invoice_status and all(invoice_status in ('invoiced', 'upselling') for invoice_status in line_invoice_status):
                order.invoice_status = 'upselling'
            else:
                order.invoice_status = 'no'

    @api.depends('transaction_ids')
    def _compute_authorized_transaction_ids(self):
        for trans in self:
            trans.authorized_transaction_ids = trans.transaction_ids.filtered(lambda t: t.state == 'authorized')

    @api.depends('transaction_ids')
    def _compute_amount_paid(self):
        """ Sum of the amount paid through all transactions for this SO. """
        for order in self:
            order.amount_paid = sum(
                tx.amount for tx in order.transaction_ids if tx.state in ('authorized', 'done')
            )

    def _compute_amount_undiscounted(self):
        for order in self:
            total = 0.0
            for line in order.order_line:
                total += (line.price_subtotal * 100)/(100-line.discount) if line.discount != 100 else (line.price_unit * line.product_uom_qty)
            order.amount_undiscounted = total

    @api.depends('order_line.customer_lead', 'date_order', 'state')
    def _compute_expected_date(self):
        """ For service and consumable, we only take the min dates. This method is extended in sale_stock to
            take the picking_policy of SO into account.
        """
        self.mapped("order_line")  # Prefetch indication
        for order in self:
            if order.state == 'cancel':
                order.expected_date = False
                continue
            dates_list = order.order_line.filtered(
                lambda line: not line.display_type and not line._is_delivery()
            ).mapped(lambda line: line and line._expected_date())
            if dates_list:
                order.expected_date = min(dates_list)
            else:
                order.expected_date = False

    def _compute_is_expired(self):
        today = fields.Date.today()
        for order in self:
            order.is_expired = order.state == 'sent' and order.validity_date and order
            .validity_date < today

    @api.depends('company_id', 'fiscal_position_id')
    def _compute_tax_country_id(self):
        for record in self:
            if record.fiscal_position_id.foreign_vat:
                record.tax_country_id = record.fiscal_position_id.country_id
            else:
                record.tax_country_id = record.company_id.account_fiscal_country_id

    @api.depends('invoice_ids.state', 'currency_id', 'amount_total')
    def _compute_amount_to_invoice(self):
        for order in self:
            # If the invoice status is 'Fully Invoiced' force the amount to invoice to equal zero and return early.
            if order.invoice_status == 'invoiced':
                order.amount_to_invoice = 0.0
                return

            order.amount_to_invoice = order.amount_total
            for invoice in order.invoice_ids.filtered(lambda x: x.state == 'posted'):
                prices = sum(invoice.line_ids.filtered(lambda x: order in x.sale_line_ids.order_id).mapped('price_total'))
                invoice_amount_currency = invoice.currency_id._convert(
                    prices * -invoice.direction_sign,
                    order.currency_id,
                    invoice.company_id,
                    invoice.date,
                )
                order.amount_to_invoice -= invoice_amount_currency

    @api.depends('amount_total', 'amount_to_invoice')
    def _compute_amount_invoiced(self):
        for order in self:
            order.amount_invoiced = order.amount_total - order.amount_to_invoice

    @api.depends('company_id', 'partner_id', 'amount_total')
    def _compute_partner_credit_warning(self):
        for order in self:
            order.with_company(order.company_id)
            order.partner_credit_warning = ''
            show_warning = order.state in ('draft', 'sent') and \
                           order.company_id.account_use_credit_limit
            if show_warning:
                order.partner_credit_warning = self.env['account.move']._build_credit_warning_message(
                    order,
                    current_amount=(order.amount_total / order.currency_rate),
                )

    @api.depends('order_line.tax_id', 'order_line.price_unit', 'amount_total', 'amount_untaxed', 'currency_id')
    def _compute_tax_totals(self):
        for order in self:
            order_lines = order.order_line.filtered(lambda x: not x.display_type)
            order.tax_totals = self.env['account.tax']._prepare_tax_totals(
                [x._convert_to_tax_base_line_dict() for x in order_lines],
                order.currency_id or order.company_id.currency_id,
            )

    @api.depends('state')
    def _compute_type_name(self):
        for record in self:
            if record.state in ('draft', 'sent', 'cancel'):
                record.type_name = _("Quotation")
            else:
                record.type_name = _("Sales Order")

    # portal.mixin override
    def _compute_access_url(self):
        super()._compute_access_url()
        for order in self:
            order.access_url = f'/my/orders/{order.id}'

    #=== CONSTRAINT METHODS ===#

    @api.constrains('company_id', 'order_line')
    def _check_order_line_company_id(self):
        for order in self:
            companies = order.order_line.product_id.company_id
            if companies and companies != order.company_id:
                bad_products = order.order_line.product_id.filtered(lambda p: p.company_id and p.company_id != order.company_id)
                raise ValidationError(_(
                    "Your quotation contains products from company %(product_company)s whereas your quotation belongs to company %(quote_company)s. \n Please change the company of your quotation or remove the products from other companies (%(bad_products)s).",
                    product_company=', '.join(companies.mapped('display_name')),
                    quote_company=order.company_id.display_name,
                    bad_products=', '.join(bad_products.mapped('display_name')),
                ))

    @api.constrains('prepayment_percent')
    def _check_prepayment_percent(self):
        for order in self:
            if order.require_payment and not (0 < order.prepayment_percent <= 1.0):
                raise ValidationError(_("Prepayment percentage must be a valid percentage."))

    #=== ONCHANGE METHODS ===#

    @api.onchange('commitment_date', 'expected_date')
    def _onchange_commitment_date(self):
        """ Warn if the commitment dates is sooner than the expected date """
        if self.commitment_date and self.expected_date and self.commitment_date < self.expected_date:
            return {
                'warning': {
                    'title': _('Requested date is too soon.'),
                    'message': _("The delivery date is sooner than the expected date."
                                 " You may be unable to honor the delivery date.")
                }
            }

    @api.onchange('company_id')
    def _onchange_company_id_warning(self):
        self.show_update_pricelist = True
        if self.order_line and self.state == 'draft':
            return {
                'warning': {
                    'title': _("Warning for the change of your quotation's company"),
                    'message': _("Changing the company of an existing quotation might need some "
                                 "manual adjustments in the details of the lines. You might "
                                 "consider updating the prices."),
                }
            }

    @api.onchange('fiscal_position_id')
    def _onchange_fpos_id_show_update_fpos(self):
        if self.order_line and (
            not self.fiscal_position_id
            or (self.fiscal_position_id and self._origin.fiscal_position_id != self.fiscal_position_id)
        ):
            self.show_update_fpos = True

    @api.onchange('partner_id')
    def _onchange_partner_id_warning(self):
        if not self.partner_id:
            return

        partner = self.partner_id

        # If partner has no warning, check its company
        if partner.sale_warn == 'no-message' and partner.parent_id:
            partner = partner.parent_id

        if partner.sale_warn and partner.sale_warn != 'no-message':
            # Block if partner only has warning but parent company is blocked
            if partner.sale_warn != 'block' and partner.parent_id and partner.parent_id.sale_warn == 'block':
                partner = partner.parent_id

            if partner.sale_warn == 'block':
                self.partner_id = False

            return {
                'warning': {
                    'title': _("Warning for %s", partner.name),
                    'message': partner.sale_warn_msg,
                }
            }

    @api.onchange('pricelist_id')
    def _onchange_pricelist_id_show_update_prices(self):
        self.show_update_pricelist = bool(self.order_line)

    @api.onchange('prepayment_percent')
    def _onchange_prepayment_percent(self):
        if not self.prepayment_percent:
            self.require_payment = False

    #=== CRUD METHODS ===#

    @api.model_create_multi
    def create(self, vals_list):
        for vals in vals_list:
            if 'company_id' in vals:
                self = self.with_company(vals['company_id'])
            if vals.get('name', _("New")) == _("New"):
                seq_date = fields.Datetime.context_timestamp(
                    self, fields.Datetime.to_datetime(vals['date_order'])
                ) if 'date_order' in vals else None
                vals['name'] = self.env['ir.sequence'].next_by_code(
                    'sale.order', sequence_date=seq_date) or _("New")

        return super().create(vals_list)

    def copy_data(self, default=None):
        if default is None:
            default = {}
        if 'order_line' not in default:
            default['order_line'] = [
                Command.create(line.copy_data()[0])
                for line in self.order_line.filtered(lambda l: not l.is_downpayment)
            ]
        return super().copy_data(default)

    @api.ondelete(at_uninstall=False)
    def _unlink_except_draft_or_cancel(self):
        for order in self:
            if order.state not in ('draft', 'cancel'):
                raise UserError(_(
                    "You can not delete a sent quotation or a confirmed sales order."
                    " You must first cancel it."))

    #=== ACTION METHODS ===#

    def action_open_discount_wizard(self):
        self.ensure_one()
        return {
            'name': _("Discount"),
            'type': 'ir.actions.act_window',
            'res_model': 'sale.order.discount',
            'view_mode': 'form',
            'target': 'new',
        }

    def action_draft(self):
        orders = self.filtered(lambda s: s.state in ['cancel', 'sent'])
        return orders.write({
            'state': 'draft',
            'signature': False,
            'signed_by': False,
            'signed_on': False,
        })

    def action_quotation_send(self):
        """ Opens a wizard to compose an email, with relevant mail template loaded by
        default """
        self.ensure_one()
        self.order_line._validate_analytic_distribution()
        lang = self.env.context.get('lang')
        mail_template = self._find_mail_template()
        if mail_template and mail_template.lang:
            lang = mail_template._render_lang(self.ids)[self.id]
        ctx = {
            'default_model': 'sale.order',
            'default_res_ids': self.ids,
            'default_template_id': mail_template.id if mail_template else None,
            'default_composition_mode': 'comment',
            'mark_so_as_sent': True,
            'default_email_layout_xmlid':
            'mail.mail_notification_layout_with_responsible_signature',
            'proforma': self.env.context.get('proforma', False),
            'force_email': True,
            'model_description': self.with_context(lang=lang).type_name,
        }
        return {
            'type': 'ir.actions.act_window',
            'view_mode': 'form',
            'res_model': 'mail.compose.message',
            'views': [(False, 'form')],
            'view_id': False,
            'target': 'new',
            'context': ctx,
        }

    def _find_mail_template(self):
        """ Get the appropriate mail template for the current sales order based on
        its state.

        If the SO is confirmed, we return the mail template for the sale confirmation.
        Otherwise, we return the quotation email template.

        :return: The correct mail template based on the current status
        :rtype: record of `mail.template` or `None` if not found
        """
        self.ensure_one()
        if self.env.context.get('proforma') or self.state != 'sale':
            return self.env.ref('sale.email_template_edi_sale', raise_if_not_found=
            False)
        else:
            return self._get_confirmation_template()

    def _get_confirmation_template(self):
        """ Get the mail template sent on SO confirmation (or for confirmed SO's).

        :return: `mail.template` record or None if default template wasn't found
        """
        self.ensure_one()
        default_confirmation_template_id = self.env['ir.config_parameter'].sudo().
        get_param(
            'sale.default_confirmation_template'
        )
```

```python
        default_confirmation_template = default_confirmation_template_id \
            and self.env['mail.template'].browse(int(default_confirmation_template_id
            )).exists()
        if default_confirmation_template:
            return default_confirmation_template
        else:
            return self.env.ref('sale.mail_template_sale_confirmation',
            raise_if_not_found=False)

    def action_quotation_sent(self):
        """ Mark the given draft quotation(s) as sent.

        :raise: UserError if any given SO is not in draft state.
        """
        if any(order.state != 'draft' for order in self):
            raise UserError(_("Only draft orders can be marked as sent directly."))

        for order in self:
            order.message_subscribe(partner_ids=order.partner_id.ids)

        self.write({'state': 'sent'})

    def action_confirm(self):
        """ Confirm the given quotation(s) and set their confirmation date.

        If the corresponding setting is enabled, also locks the Sale Order.

        :return: True
        :rtype: bool
        :raise: UserError if trying to confirm cancelled SO's
        """
        if not all(order._can_be_confirmed() for order in self):
            raise UserError(_(
                "The following orders are not in a state requiring confirmation: %s",
                ", ".join(self.mapped('display_name')),
            ))

        self.order_line._validate_analytic_distribution()

        for order in self:
            order.validate_taxes_on_sales_order()
            if order.partner_id in order.message_partner_ids:
                continue
            order.message_subscribe([order.partner_id.id])

        self.write(self._prepare_confirmation_values())

        # Context key 'default_name' is sometimes propagated up to here.
        # We don't need it and it creates issues in the creation of linked records.
        context = self._context.copy()
        context.pop('default_name', None)

        self.with_context(context)._action_confirm()
        if self.env.user.has_group('sale.group_auto_done_setting'):
            self.action_lock()

        return True

    def _can_be_confirmed(self):
        self.ensure_one()
        return self.state in {'draft', 'sent'}

    def _prepare_confirmation_values(self):
        """ Prepare the sales order confirmation values.

        Note: self can contain multiple records.

        :return: Sales Order confirmation values
        :rtype: dict
        """
        return {
            'state': 'sale',
            'date_order': fields.Datetime.now()
        }

    def _action_confirm(self):
        """ Implementation of additional mechanism of Sales Order confirmation.
            This method should be extended when the confirmation should generated
            other documents. In this method, the SO are in 'sale' state (not yet 'done').
        """
        # create an analytic account if at least an expense product
        for order in self:
            if any(expense_policy not in [False, 'no'] for expense_policy in order.order_line.product_id.mapped('expense_policy')):
                if not order.analytic_account_id:
                    order._create_analytic_account()

    def _send_order_confirmation_mail(self):
        """ Send a mail to the SO customer to inform them that their order has been confirmed.

        :return: None
        """
        for order in self:
            mail_template = order._get_confirmation_template()
            order._send_order_notification_mail(mail_template)

    def _send_payment_succeeded_for_order_mail(self):
        """ Send a mail to the SO customer to inform them that a payment has been initiated.

        :return: None
        """
        mail_template = self.env.ref(
            'sale.mail_template_sale_payment_executed', raise_if_not_found=False
        )
        for order in self:
            order._send_order_notification_mail(mail_template)

    def _send_order_notification_mail(self, mail_template):
        """ Send a mail to the customer

        Note: self.ensure_one()

        :param mail.template mail_template: the template used to generate the mail
        :return: None
        """
        self.ensure_one()

        if not mail_template:
            return

        if self.env.su:
            # sending mail in sudo was meant for it being sent from superuser
            self = self.with_user(SUPERUSER_ID)

        self.with_context(force_send=True).message_post_with_source(
            mail_template,
            email_layout_xmlid='mail.mail_notification_layout_with_responsible_signature',
            subtype_xmlid='mail.mt_comment',
        )

    def action_lock(self):
        for order in self:
            tx = order.sudo().transaction_ids._get_last()
            if tx and tx.state == 'pending' and tx.provider_id.code == 'custom' and tx.provider_id.custom_mode == 'wire_transfer':
                tx._set_done()
                tx.write({'is_post_processed': True})
        self.locked = True

    def action_unlock(self):
            self.locked = False

    def action_cancel(self):
        """ Cancel SO after showing the cancel wizard when needed. (cfr
        :meth:`_show_cancel_wizard`)

        For post-cancel operations, please only override :meth:`_action_cancel`.

        note: self.ensure_one() if the wizard is shown.
        """
        if any(order.locked for order in self):
            raise UserError(_("You cannot cancel a locked order. Please unlock it
            first."))
        cancel_warning = self._show_cancel_wizard()
        if cancel_warning:
            self.ensure_one()
            template_id = self.env['ir.model.data']._xmlid_to_res_id(
                'sale.mail_template_sale_cancellation', raise_if_not_found=False
            )
            lang = self.env.context.get('lang')
            template = self.env['mail.template'].browse(template_id)
            if template.lang:
                lang = template._render_lang(self.ids)[self.id]
            ctx = {
                'default_template_id': template_id,
                'default_order_id': self.id,
                'mark_so_as_canceled': True,
                'default_email_layout_xmlid':
                "mail.mail_notification_layout_with_responsible_signature",
                'model_description': self.with_context(lang=lang).type_name,
            }
            return {
                'name': _('Cancel %s', self.type_name),
                'view_mode': 'form',
                'res_model': 'sale.order.cancel',
                'view_id': self.env.ref('sale.sale_order_cancel_view_form').id,
                'type': 'ir.actions.act_window',
                'context': ctx,
                'target': 'new'
            }
        else:
            return self._action_cancel()

    def _action_cancel(self):
        inv = self.invoice_ids.filtered(lambda inv: inv.state == 'draft')
        inv.button_cancel()
        return self.write({'state': 'cancel'})

    def _show_cancel_wizard(self):
        """ Decide whether the sale.order.cancel wizard should be shown to cancel
        specified orders.

        :return: True if there is any non-draft order in the given orders
        :rtype: bool
        """
        if self.env.context.get('disable_cancel_warning'):
            return False
        return any(so.state != 'draft' for so in self)

    def action_preview_sale_order(self):
        self.ensure_one()
        return {
            'type': 'ir.actions.act_url',
            'target': 'self',
            'url': self.get_portal_url(),
        }

    def action_update_taxes(self):
        self.ensure_one()

        self._recompute_taxes()

        if self.partner_id:
            self.message_post(body=_("Product taxes have been recomputed according to fiscal position %s.",
                self.fiscal_position_id._get_html_link() if self.fiscal_position_id else "")
            )

    def _recompute_taxes(self):
        lines_to_recompute = self.order_line.filtered(lambda line: not line.display_type)
        lines_to_recompute._compute_tax_id()
        self.show_update_fpos = False

    def action_update_prices(self):
        self.ensure_one()

        self._recompute_prices()

        if self.pricelist_id:
            message = _("Product prices have been recomputed according to pricelist %s.",
                self.pricelist_id._get_html_link())
        else:
            message = _("Product prices have been recomputed.")
        self.message_post(body=message)

    def _recompute_prices(self):
        lines_to_recompute = self._get_update_prices_lines()
        lines_to_recompute.invalidate_recordset(['pricelist_item_id'])
        lines_to_recompute._compute_price_unit()
        # Special case: we want to overwrite the existing discount on _recompute_prices call
        # i.e. to make sure the discount is correctly reset
        # if pricelist discount_policy is different than when the price was first computed.
        lines_to_recompute.discount = 0.0
        lines_to_recompute._compute_discount()
        self.show_update_pricelist = False

    def _default_order_line_values(self):
        default_data = super()._default_order_line_values()
        new_default_data = self.env['sale.order.line']._get_product_catalog_lines_data()
        return {**default_data, **new_default_data}

    def _get_action_add_from_catalog_extra_context(self):
        return {
            **super()._get_action_add_from_catalog_extra_context(),
            'product_catalog_currency_id': self.currency_id.id,
            'product_catalog_digits': self.order_line._fields['price_unit'].get_digits(self.env),
        }

    def _get_product_catalog_domain(self):
        return expression.AND([super()._get_product_catalog_domain(), [('sale_ok', '=', True)]])

    # INVOICING #

    def _prepare_invoice(self):
        """
        Prepare the dict of values to create the new invoice for a sales order. This method may be
        overridden to implement custom invoice generation (making sure to call super() to establish
        a clean extension chain).
        """
        self.ensure_one()

        values = {
            'ref': self.client_order_ref or '',
            'move_type': 'out_invoice',
            'narration': self.note,
            'currency_id': self.currency_id.id,
            'campaign_id': self.campaign_id.id,
            'medium_id': self.medium_id.id,
            'source_id': self.source_id.id,
            'team_id': self.team_id.id,
            'partner_id': self.partner_invoice_id.id,
            'partner_shipping_id': self.partner_shipping_id.id,
            'fiscal_position_id': (self.fiscal_position_id or self.fiscal_position_id._get_fiscal_position(self.partner_invoice_id)).id,
            'invoice_origin': self.name,
            'invoice_payment_term_id': self.payment_term_id.id,
            'invoice_user_id': self.user_id.id,
            'payment_reference': self.reference,
            'transaction_ids': [Command.set(self.transaction_ids.ids)],
            'company_id': self.company_id.id,
            'invoice_line_ids': [],
            'user_id': self.user_id.id,
        }
        if self.journal_id:
            values['journal_id'] = self.journal_id.id
        return values

    def action_view_invoice(self, invoices=False):
        if not invoices:
            invoices = self.mapped('invoice_ids')
        action = self.env['ir.actions.actions']._for_xml_id('account.action_move_out_invoice_type')
        if len(invoices) > 1:
            action['domain'] = [('id', 'in', invoices.ids)]
        elif len(invoices) == 1:
            form_view = [(self.env.ref('account.view_move_form').id, 'form')]
            if 'views' in action:
                action['views'] = form_view + [(state,view) for state,view in action['views'] if view != 'form']
            else:
                action['views'] = form_view
            action['res_id'] = invoices.id
        else:
            action = {'type': 'ir.actions.act_window_close'}

        context = {
            'default_move_type': 'out_invoice',
        }
        if len(self) == 1:
            context.update({
                'default_partner_id': self.partner_id.id,
                'default_partner_shipping_id': self.partner_shipping_id.id,
                'default_invoice_payment_term_id': self.payment_term_id.id or self.partner_id.property_payment_term_id.id or self.env['account.move'].default_get(['invoice_payment_term_id']).get('invoice_payment_term_id'),
                'default_invoice_origin': self.name,
            })
        action['context'] = context
        return action

    def _get_invoice_grouping_keys(self):
        return ['company_id', 'partner_id', 'currency_id']

    def _nothing_to_invoice_error_message(self):
        return _(
            "Cannot create an invoice. No items are available to invoice.\n\n"
            "To resolve this issue, please ensure that:\n"
            "   \u2022 The products have been delivered before attempting to invoice them.\n"
            "   \u2022 The invoicing policy of the product is configured correctly.\n\n"
            "If you want to invoice based on ordered quantities instead:\n"
            "   \u2022 For consumable or storable products, open the product, go to the 'General Information' tab and change the 'Invoicing Policy' from 'Delivered Quantities' to 'Ordered Quantities'.\n"
            "   \u2022 For services (and other products), change the 'Invoicing Policy' to 'Prepaid/Fixed Price'.\n"
        )

    def _get_update_prices_lines(self):
        """ Hook to exclude specific lines which should not be updated based on price list recomputation """
        return self.order_line.filtered(lambda line: not line.display_type)

    def _get_invoiceable_lines(self, final=False):
        """Return the invoiceable lines for order `self`."""
        down_payment_line_ids = []
        invoiceable_line_ids = []
        pending_section = None
        precision = self.env['decimal.precision'].precision_get('Product Unit of Measure')

        for line in self.order_line:
            if line.display_type == 'line_section':
                # Only invoice the section if one of its lines is invoiceable
                pending_section = line
                continue
            if line.display_type != 'line_note' and float_is_zero(line.qty_to_invoice, precision_digits=precision):
                continue
            if line.qty_to_invoice > 0 or (line.qty_to_invoice < 0 and final) or line.display_type == 'line_note':
                if line.is_downpayment:
                    # Keep down payment lines separately, to put them together
                    # at the end of the invoice, in a specific dedicated section.
                    down_payment_line_ids.append(line.id)
                    continue
                if pending_section:
                    invoiceable_line_ids.append(pending_section.id)
                    pending_section = None
                invoiceable_line_ids.append(line.id)

        return self.env['sale.order.line'].browse(invoiceable_line_ids + down_payment_line_ids)

    def _create_invoices(self, grouped=False, final=False, date=None):
        """ Create invoice(s) for the given Sales Order(s).

        :param bool grouped: if True, invoices are grouped by SO id.
            If False, invoices are grouped by keys returned by :meth:`_get_invoice_grouping_keys`
        :param bool final: if True, refunds will be generated if necessary
        :param date: unused parameter
        :returns: created invoices
        :rtype: `account.move` recordset
        :raises: UserError if one of the orders has no invoiceable lines.
        """
        if not self.env['account.move'].check_access_rights('create', False):
            try:
                self.check_access_rights('write')
                self.check_access_rule('write')
            except AccessError:
                return self.env['account.move']

        # 1) Create invoices.
        invoice_vals_list = []
        invoice_item_sequence = 0 # Incremental sequencing to keep the lines order on the invoice.
        for order in self:
            order = order.with_company(order.company_id).with_context(lang=order.partner_invoice_id.lang)

            invoice_vals = order._prepare_invoice()
            invoiceable_lines = order._get_invoiceable_lines(final)

            if not any(not line.display_type for line in invoiceable_lines):
                continue
            invoice_line_vals = []
            down_payment_section_added = False
            for line in invoiceable_lines:
                if not down_payment_section_added and line.is_downpayment:
                    # Create a dedicated section for the down payments
                    # (put at the end of the invoiceable_lines)
                    invoice_line_vals.append(
                        Command.create(
                            order._prepare_down_payment_section_line(sequence=
                            invoice_item_sequence)
                        ),
                    )
                    down_payment_section_added = True
                    invoice_item_sequence += 1
                invoice_line_vals.append(
                    Command.create(
                        line._prepare_invoice_line(sequence=invoice_item_sequence)
                    ),
                )
                invoice_item_sequence += 1

            invoice_vals['invoice_line_ids'] += invoice_line_vals
            invoice_vals_list.append(invoice_vals)

        if not invoice_vals_list and self._context.get('raise_if_nothing_to_invoice',
        True):
            raise UserError(self._nothing_to_invoice_error_message())

        # 2) Manage 'grouped' parameter: group by (partner_id, currency_id).
        if not grouped:
            new_invoice_vals_list = []
            invoice_grouping_keys = self._get_invoice_grouping_keys()
            invoice_vals_list = sorted(
                invoice_vals_list,
                key=lambda x: [
                    x.get(grouping_key) for grouping_key in invoice_grouping_keys
                ]
            )
            for _grouping_keys, invoices in groupby(invoice_vals_list, key=lambda x: [
            x.get(grouping_key) for grouping_key in invoice_grouping_keys]):
                origins = set()
                payment_refs = set()
                refs = set()
                ref_invoice_vals = None
                for invoice_vals in invoices:
                    if not ref_invoice_vals:
                        ref_invoice_vals = invoice_vals
                    else:
                        ref_invoice_vals['invoice_line_ids'] += invoice_vals[
                        'invoice_line_ids']
                    origins.add(invoice_vals['invoice_origin'])
                    payment_refs.add(invoice_vals['payment_reference'])
                    refs.add(invoice_vals['ref'])
                ref_invoice_vals.update({
                    'ref': ', '.join(refs)[:2000],
                    'invoice_origin': ', '.join(origins),
                    'payment_reference': len(payment_refs) == 1 and payment_refs.pop()
                     or False,
                })
                new_invoice_vals_list.append(ref_invoice_vals)
            invoice_vals_list = new_invoice_vals_list

        # 3) Create invoices.

        # As part of the invoice creation, we make sure the sequence of multiple SO
        do not interfere
        # in a single invoice. Example:
        # SO 1:
        # - Section A (sequence: 10)
        # - Product A (sequence: 11)
        # SO 2:
```

```python
        # - Section B (sequence: 10)
        # - Product B (sequence: 11)
        #
        # If SO 1 & 2 are grouped in the same invoice, the result will be:
        # - Section A (sequence: 10)
        # - Section B (sequence: 10)
        # - Product A (sequence: 11)
        # - Product B (sequence: 11)
        #
        # Resequencing should be safe, however we resequence only if there are less invoices than
        # orders, meaning a grouping might have been done. This could also mean that only a part
        # of the selected SO are invoiceable, but resequencing in this case shouldn't be an issue.
        if len(invoice_vals_list) < len(self):
            SaleOrderLine = self.env['sale.order.line']
            for invoice in invoice_vals_list:
                sequence = 1
                for line in invoice['invoice_line_ids']:
                    line[2]['sequence'] = SaleOrderLine._get_invoice_line_sequence(new=sequence, old=line[2]['sequence'])
                    sequence += 1

        # Manage the creation of invoices in sudo because a salesperson must be able to generate an invoice from a
        # sale order without "billing" access rights. However, he should not be able to create an invoice from scratch.
        moves = self.env['account.move'].sudo().with_context(default_move_type='out_invoice').create(invoice_vals_list)

        # 4) Some moves might actually be refunds: convert them if the total amount is negative
        # We do this after the moves have been created since we need taxes, etc. to know if the total
        # is actually negative or not
        if final:
            moves.sudo().filtered(lambda m: m.amount_total < 0).action_switch_move_type()
        for move in moves:
            if final:
                # Downpayment might have been determined by a fixed amount set by the user.
                # This amount is tax included. This can lead to rounding issues.
                # E.g. a user wants a 100€ DP on a product with 21% tax.
                # 100 / 1.21 = 82.64, 82.64 * 1,21 = 99.99
                # This is already corrected by adding/removing the missing cents on the DP invoice,
                # but must also be accounted for on the final invoice.

                delta_amount = 0
                for order_line in self.order_line:
                    if not order_line.is_downpayment:
                        continue
                    inv_amt = order_amt = 0
                    for invoice_line in order_line.invoice_lines:
                        if invoice_line.move_id == move:
                            inv_amt += invoice_line.price_total
                        elif invoice_line.move_id.state != 'cancel':  # filter out canceled dp lines
                            order_amt += invoice_line.price_total
                    if inv_amt and order_amt:
                        # if not inv_amt, this order line is not related to current move
                        # if no order_amt, dp order line was not invoiced
                        delta_amount += (inv_amt * (1 if move.is_inbound() else -1)) + order_amt

                if not move.currency_id.is_zero(delta_amount):
                    receivable_line = move.line_ids.filtered(
                        lambda aml: aml.account_id.account_type == 'asset_receivable')[:1]
                    product_lines = move.line_ids.filtered(
                        lambda aml: aml.display_type == 'product' and aml.is_downpayment)
                    tax_lines = move.line_ids.filtered(
                        lambda aml: aml.tax_line_id.amount_type not in (False, 'fixed'))
                    if tax_lines and product_lines and receivable_line:
                        line_commands = [Command.update(receivable_line.id, {
                            'amount_currency': receivable_line.amount_currency + delta_amount,
                        })]
                        delta_sign = 1 if delta_amount > 0 else -1
                        for lines, attr, sign in (
                            (product_lines, 'price_total', -1 if move.is_inbound() else 1),
                            (tax_lines, 'amount_currency', 1),
                        ):
                            remaining = delta_amount
                            lines_len = len(lines)
                            for line in lines:
                                if move.currency_id.compare_amounts(remaining, 0) != delta_sign:
                                    break
                                amt = delta_sign * max(
                                    move.currency_id.rounding,
                                    abs(move.currency_id.round(remaining / lines_len)),
                                )
                                remaining -= amt
                                line_commands.append(Command.update(line.id, {attr: line[attr] + amt * sign}))
                        move.line_ids = line_commands

            move.message_post_with_source(
                'mail.message_origin_link',
                render_values={'self': move, 'origin': move.line_ids.sale_line_ids.order_id},
                subtype_xmlid='mail.mt_note',
            )
        return moves

    # MAIL #

    def _track_finalize(self):
        """ Override of `mail` to prevent logging changes when the SO is in a draft state. """
        if (len(self) == 1
            # The method _track_finalize is sometimes called too early or too late and it
            # might cause a desynchronization with the cache, thus this condition is needed.
            and self.env.cache.contains(self, self._fields['state']) and self.state == 'draft'):
            self.env.cr.precommit.data.pop(f'mail.tracking.{self._name}', {})
            self.env.flush_all()
            return
        return super()._track_finalize()

    @api.returns('mail.message', lambda value: value.id)
    def message_post(self, **kwargs):
        if self.env.context.get('mark_so_as_sent'):
            self.filtered(lambda o: o.state == 'draft').with_context(tracking_disable=True).write({'state': 'sent'})
        so_ctx = {'mail_post_autofollow': self.env.context.get('mail_post_autofollow', True)}
        if self.env.context.get('mark_so_as_sent') and 'mail_notify_author' not in kwargs:
            kwargs['notify_author'] = self.env.user.partner_id.id in (kwargs.get('partner_ids') or [])
        return super(SaleOrder, self.with_context(**so_ctx)).message_post(**kwargs)

    def _notify_get_recipients_groups(self, message, model_description, msg_vals=None
    ):
        """ Give access button to users and portal customer as portal is integrated
        in sale. Customer and portal group have probably no right to see
        the document so they don't have the access button. """
        groups = super()._notify_get_recipients_groups(
            message, model_description, msg_vals=msg_vals
        )
        if not self:
            return groups

        self.ensure_one()
        if self._context.get('proforma'):
            for group in [g for g in groups if g[0] in ('portal_customer', 'portal', 'follower', 'customer')]:
                group[2]['has_button_access'] = False
            return groups
        local_msg_vals = dict(msg_vals or {})

        # portal customers have full access (existence not granted, depending on partner_id)
        try:
            customer_portal_group = next(group for group in groups if group[0] == 'portal_customer')
        except StopIteration:
            pass
        else:
            access_opt = customer_portal_group[2].setdefault('button_access', {})
            is_tx_pending = self.get_portal_last_transaction().state == 'pending'
            if self._has_to_be_signed():
                if self._has_to_be_paid():
                    access_opt['title'] = _("View Quotation") if is_tx_pending else _("Sign & Pay Quotation")
                else:
                    access_opt['title'] = _("Accept & Sign Quotation")
            elif self._has_to_be_paid() and not is_tx_pending:
                access_opt['title'] = _("Accept & Pay Quotation")
            elif self.state in ('draft', 'sent'):
                access_opt['title'] = _("View Quotation")

        # enable followers that have access through portal
        follower_group = next(group for group in groups if group[0] == 'follower')
        follower_group[2]['active'] = True
        follower_group[2]['has_button_access'] = True
        access_opt = follower_group[2].setdefault('button_access', {})
        if self.state in ('draft', 'sent'):
            access_opt['title'] = _("View Quotation")
        else:
            access_opt['title'] = _("View Order")
        access_opt['url'] = self._notify_get_action_link('view', **local_msg_vals)

        return groups

    def _notify_by_email_prepare_rendering_context(self, message, msg_vals, model_description=False,
                                                   force_email_company=False,
                                                   force_email_lang=False):
        render_context = super()._notify_by_email_prepare_rendering_context(
            message, msg_vals, model_description=model_description,
            force_email_company=force_email_company, force_email_lang=force_email_lang
        )
        lang_code = render_context.get('lang')
        subtitles = [
            render_context['record'].name,
        ]

        if self.amount_total:
            # Do not show the price in subtitles if zero (e.g. e-commerce orders are created empty)
            subtitles.append(
                format_amount(self.env, self.amount_total, self.currency_id, lang_code=lang_code),
            )
```

```python

        if self.validity_date and self.state in ['draft', 'sent']:
            formatted_date = format_date(self.env, self.validity_date, lang_code=
            lang_code)
            subtitles.append(_("Expires on %(date)s", date=formatted_date))

        render_context['subtitles'] = subtitles
        return render_context

    def _phone_get_number_fields(self):
        """ No phone or mobile field is available on sale model. Instead SMS will
        fallback on partner-based computation using ``_mail_get_partner_fields``. """
        return []

    def _track_subtype(self, init_values):
        self.ensure_one()
        if 'state' in init_values and self.state == 'sale':
            return self.env.ref('sale.mt_order_confirmed')
        elif 'state' in init_values and self.state == 'sent':
            return self.env.ref('sale.mt_order_sent')
        return super()._track_subtype(init_values)

    # PAYMENT #

    def _force_lines_to_invoice_policy_order(self):
        """Force the qty_to_invoice to be computed as if the invoice_policy
        was set to "Ordered quantities", independently of the product configuration.

        This is needed for the automatic invoice logic, as we want to automatically
        invoice the full SO when it's paid.
        """
        for line in self.order_line:
            if line.state == 'sale':
                # No need to set 0 as it is already the standard logic in the compute
                method.
                line.qty_to_invoice = line.product_uom_qty - line.qty_invoiced

    def payment_action_capture(self):
        """ Capture all transactions linked to this sale order. """
        self.ensure_one()
        payment_utils.check_rights_on_recordset(self)

        # In sudo mode to bypass the checks on the rights on the transactions.
        return self.transaction_ids.sudo().action_capture()

    def payment_action_void(self):
        """ Void all transactions linked to this sale order. """
        payment_utils.check_rights_on_recordset(self)

        # In sudo mode to bypass the checks on the rights on the transactions.
        self.authorized_transaction_ids.sudo().action_void()

    def get_portal_last_transaction(self):
        self.ensure_one()
        return self.transaction_ids.sudo()._get_last()

    def _get_order_lines_to_report(self):
        down_payment_lines = self.order_line.filtered(lambda line:
            line.is_downpayment
            and not line.display_type
            and not line._get_downpayment_state()
        )

        def show_line(line):
            if not line.is_downpayment:
                return True
            elif line.display_type and down_payment_lines:
                return True  # Only show the down payment section if down payments
                were posted
            elif line in down_payment_lines:
                return True  # Only show posted down payments
            else:
                return False

        return self.order_line.filtered(show_line)

    def _get_default_payment_link_values(self):
        self.ensure_one()
        amount_max = self.amount_total - self.amount_paid

        # Always default to the minimum value needed to confirm the order:
        # - order is not confirmed yet
        # - can be confirmed online
        # - we have still not paid enough for confirmation.
        prepayment_amount = self._get_prepayment_required_amount()
        if (
            self.state in ('draft', 'sent')
            and self.require_payment
            and self.currency_id.compare_amounts(prepayment_amount, self.amount_paid)
            > 0
        ):
            amount = prepayment_amount - self.amount_paid
        else:
            amount = amount_max

        return {
            'currency_id': self.currency_id.id,
            'partner_id': self.partner_invoice_id.id,
            'amount': amount,
            'amount_max': amount_max,
            'amount_paid': self.amount_paid,
        }

    # PORTAL #

    def _has_to_be_signed(self):
        """A sale order has to be signed when:
        - its state is 'draft' or `sent`
        - it's not expired;
        - it requires a signature;
        - it's not already signed.

        Note: self.ensure_one()

        :return: Whether the sale order has to be signed.
        :rtype: bool
        """
        self.ensure_one()
        return (
            self.state in ['draft', 'sent']
            and not self.is_expired
            and self.require_signature
            and not self.signature
        )

    def _has_to_be_paid(self):
        """A sale order has to be paid when:
        - its state is 'draft' or `sent`;
        - it's not expired;
        - it requires a payment;
        - the last transaction's state isn't `done`;
        - the total amount is strictly positive.

        Note: self.ensure_one()

        :return: Whether the sale order has to be paid.
        :rtype: bool
        """
        self.ensure_one()
        transaction = self.get_portal_last_transaction()
        return (
            self.state in ['draft', 'sent']
            and not self.is_expired
            and self.require_payment
```

```python
                and transaction.state != 'done'
                and self.amount_total > 0
            )

    def _get_portal_return_action(self):
        """ Return the action used to display orders when returning from customer
        portal. """
        self.ensure_one()
        return self.env.ref('sale.action_quotations_with_onboarding')

    def _get_name_portal_content_view(self):
        """ This method can be inherited by localizations who want to localize the
        online quotation view. """
        self.ensure_one()
        return 'sale.sale_order_portal_content'

    def _get_name_tax_totals_view(self):
        """ This method can be inherited by localizations who want to localize the
        taxes displayed on the portal and sale order report. """
        return 'sale.document_tax_totals'

    def _get_report_base_filename(self):
        self.ensure_one()
        return f'{self.type_name} {self.name}'

    #=== CORE METHODS OVERRIDES ===#

    @api.model
    def get_empty_list_help(self, help_msg):
        self = self.with_context(
            empty_list_help_document_name=_("sale order"),
        )
        return super().get_empty_list_help(help_msg)

    def _compute_field_value(self, field):
        if field.name != 'invoice_status' or self.env.context.get(
        'mail_activity_automation_skip'):
            return super()._compute_field_value(field)

        filtered_self = self.filtered(
            lambda so: so.ids
                and (so.user_id or so.partner_id.user_id)
                and so._origin.invoice_status != 'upselling')
        super()._compute_field_value(field)

        upselling_orders = filtered_self.filtered(lambda so: so.invoice_status ==
        'upselling')
        upselling_orders._create_upsell_activity()

    #=== BUSINESS METHODS ===#

    def _create_upsell_activity(self):
        if not self:
            return

        self.activity_unlink(['sale.mail_act_sale_upsell'])
        for order in self:
            order_ref = order._get_html_link()
            customer_ref = order.partner_id._get_html_link()
            order.activity_schedule(
                'sale.mail_act_sale_upsell',
                user_id=order.user_id.id or order.partner_id.user_id.id,
                note=_("Upsell %(order)s for customer %(customer)s", order=order_ref,
                customer=customer_ref))

    def _prepare_analytic_account_data(self, prefix=None):
        """ Prepare SO analytic account creation values.

        :param str prefix: The prefix of the to-be-created analytic account name
        :return: `account.analytic.account` creation values
        :rtype: dict
        """
        self.ensure_one()
        name = self.name
        if prefix:
            name = prefix + ": " + self.name
        plan = self.env['account.analytic.plan'].sudo().search([], limit=1)
        if not plan:
            plan = self.env['account.analytic.plan'].sudo().create({
                'name': 'Default',
            })
        return {
            'name': name,
            'code': self.client_order_ref,
            'company_id': self.company_id.id,
            'plan_id': plan.id,
            'partner_id': self.partner_id.id,
        }

    def _create_analytic_account(self, prefix=None):
        """ Create a new analytic account for the given orders.

        :param str prefix: if specified, the account name will be '<prefix>:
        <so_reference>'.
            If not, the account name will be the Sales Order reference.
        :return: None
        """
        for order in self:
            analytic = self.env['account.analytic.account'].create(order.
            _prepare_analytic_account_data(prefix))
            order.analytic_account_id = analytic

    def _prepare_down_payment_section_line(self, **optional_values):
        """ Prepare the values to create a new down payment section.

        :param dict optional_values: any parameter that should be added to the
        returned down payment section
        :return: `account.move.line` creation values
        :rtype: dict
        """
        self.ensure_one()
        context = {'lang': self.partner_id.lang}
        down_payments_section_line = {
            'display_type': 'line_section',
            'name': _("Down Payments"),
            'product_id': False,
            'product_uom_id': False,
            'quantity': 0,
            'discount': 0,
            'price_unit': 0,
            'account_id': False,
            **optional_values
        }
        del context
        return down_payments_section_line

    def _get_prepayment_required_amount(self):
        """ Return the minimum amount needed to confirm automatically the quotation.

        Note: self.ensure_one()

        :return: The minimum amount needed to confirm automatically the quotation.
        :rtype: float
        """
        self.ensure_one()
        if self.prepayment_percent == 1.0 or not self.require_payment:
            return self.amount_total
        else:
            return self.currency_id.round(self.amount_total * self.prepayment_percent)

    def _is_confirmation_amount_reached(self):
        """ Return whether `self.amount_paid` is higher than the prepayment required
        amount.

        Note: self.ensure_one()

        :return: Whether `self.amount_paid` is higher than the prepayment required
        amount.
        :rtype: bool
        """
        self.ensure_one()
        amount_comparison = self.currency_id.compare_amounts(
            self._get_prepayment_required_amount(), self.amount_paid,
        )
        return amount_comparison <= 0

    def _generate_downpayment_invoices(self):
        """ Generate invoices as down payments for sale order.

        :return: The generated down payment invoices.
        :rtype: recordset of `account.move`
        """
        generated_invoices = self.env['account.move']

        for order in self:
            downpayment_wizard = order.env['sale.advance.payment.inv'].create({
                'sale_order_ids': order,
                'advance_payment_method': 'fixed',
                'fixed_amount': order.amount_paid,
            })
            generated_invoices |= downpayment_wizard._create_invoices(order)

        return generated_invoices

    def _get_product_catalog_order_data(self, products, **kwargs):
        pricelist = self.pricelist_id._get_products_price(
            quantity=1.0,
            products=products,
            currency=self.currency_id,
            date=self.date_order,
            **kwargs,
        )
        return {product_id: {'price': price} for product_id, price in pricelist.items
        ()}

    def _get_product_catalog_record_lines(self, product_ids):
        grouped_lines = defaultdict(lambda: self.env['sale.order.line'])
        for line in self.order_line:
            if line.display_type or line.product_id.id not in product_ids:
                continue
            grouped_lines[line.product_id] |= line
        return grouped_lines

    def _get_product_documents(self):
        self.ensure_one()

        documents = (
            self.order_line.product_id.product_document_ids
            | self.order_line.product_template_id.product_document_ids
        )
        return self._filter_product_documents(documents).sorted()

    def _filter_product_documents(self, documents):
        return documents.filtered(
            lambda document:
                document.attached_on == 'quotation'
                or (self.state == 'sale' and document.attached_on == 'sale_order')
        )

    def _update_order_line_info(self, product_id, quantity, **kwargs):
        """ Update sale order line information for a given product or create a
        new one if none exists yet.
        :param int product_id: The product, as a `product.product` id.
        :return: The unit price of the product, based on the pricelist of the
                 sale order and the quantity selected.
        :rtype: float
        """
        sol = self.order_line.filtered(lambda line: line.product_id.id == product_id)
        if sol:
            if quantity != 0:
                sol.product_uom_qty = quantity
            elif self.state in ['draft', 'sent']:
                price_unit = self.pricelist_id._get_product_price(
                    product=sol.product_id,
                    quantity=1.0,
                    currency=self.currency_id,
                    date=self.date_order,
                    **kwargs,
                )
                sol.unlink()
                return price_unit
            else:
                sol.product_uom_qty = 0
        elif quantity > 0:
            sol = self.env['sale.order.line'].create({
                'order_id': self.id,
                'product_id': product_id,
                'product_uom_qty': quantity,
                'sequence': ((self.order_line and self.order_line[-1].sequence + 1) or 10),  # put it at the end of the order
            })
        return sol.price_unit

    #=== HOOKS ===#

    def add_option_to_order_with_taxcloud(self):
        self.ensure_one()

    def validate_taxes_on_sales_order(self):
        # Override for correct taxcloud computation
        # when using coupon and delivery
        return True

    #=== TOOLING ===#

    def _is_readonly(self):
        """ Return Whether the sale order is read-only or not based on the state or the lock status.

        A sale order is considered read-only if its state is 'cancel' or if the sale order is
        locked.

        :return: Whether the sale order is read-only or not.
        :rtype: bool
        """
        self.ensure_one()
        return self.state == 'cancel' or self.locked

    def _is_paid(self):
        """ Return whether the sale order is paid or not based on the linked transactions.

        A sale order is considered paid if the sum of all the linked transaction is equal to or
        higher than `self.amount_total`.

        :return: Whether the sale order is paid or not.
        :rtype: bool
        """
        self.ensure_one()
        return self.currency_id.compare_amounts(self.amount_paid, self.amount_total) >= 0

# sale_order_line.py
 # -*- coding: utf-8 -*-
# Part of Odoo. See LICENSE file for full copyright and licensing details.

```

```python
from collections import defaultdict
from datetime import timedelta
from markupsafe import Markup

from odoo import api, fields, models, _
from odoo.exceptions import UserError
from odoo.fields import Command
from odoo.osv import expression
from odoo.tools import float_is_zero, float_compare, float_round, format_date, groupby


class SaleOrderLine(models.Model):
    _name = 'sale.order.line'
    _inherit = 'analytic.mixin'
    _description = "Sales Order Line"
    _rec_names_search = ['name', 'order_id.name']
    _order = 'order_id, sequence, id'
    _check_company_auto = True

    _sql_constraints = [
        ('accountable_required_fields',
            "CHECK(display_type IS NOT NULL OR (product_id IS NOT NULL AND
            product_uom IS NOT NULL))",
            "Missing required fields on accountable sale order line."),
        ('non_accountable_null_fields',
            "CHECK(display_type IS NULL OR (product_id IS NULL AND price_unit = 0 AND
            product_uom_qty = 0 AND product_uom IS NULL AND customer_lead = 0))",
            "Forbidden values on non-accountable sale order line"),
    ]

    # Fields are ordered according by tech & business logics
    # and computed fields are defined after their dependencies.
    # This reduces execution stacks depth when precomputing fields
    # on record creation (and is also a good ordering logic imho)

    order_id = fields.Many2one(
        comodel_name='sale.order',
        string="Order Reference",
        required=True, ondelete='cascade', index=True, copy=False)
    sequence = fields.Integer(string="Sequence", default=10)

    # Order-related fields
    company_id = fields.Many2one(
        related='order_id.company_id',
        store=True, index=True, precompute=True)
    currency_id = fields.Many2one(
        related='order_id.currency_id',
        depends=['order_id.currency_id'],
        store=True, precompute=True)
    order_partner_id = fields.Many2one(
        related='order_id.partner_id',
        string="Customer",
        store=True, index=True, precompute=True)
    salesman_id = fields.Many2one(
        related='order_id.user_id',
        string="Salesperson",
        store=True, precompute=True)
    state = fields.Selection(
        related='order_id.state',
        string="Order Status",
        copy=False, store=True, precompute=True)
    tax_country_id = fields.Many2one(related='order_id.tax_country_id')

    # Fields specifying custom line logic
    display_type = fields.Selection(
        selection=[
            ('line_section', "Section"),
            ('line_note', "Note"),
        ],
        default=False)
    is_downpayment = fields.Boolean(
        string="Is a down payment",
        help="Down payments are made when creating invoices from a sales order."
            " They are not copied when duplicating a sales order.")
    is_expense = fields.Boolean(
        string="Is expense",
        help="Is true if the sales order line comes from an expense or a vendor bills"
        )

    # Generic configuration fields
    product_id = fields.Many2one(
        comodel_name='product.product',
        string="Product",
        change_default=True, ondelete='restrict', check_company=True, index=
        'btree_not_null',
        domain="[('sale_ok', '=', True)]")
    product_template_id = fields.Many2one(
        string="Product Template",
        comodel_name='product.template',
        compute='_compute_product_template_id',
        readonly=False,
        search='_search_product_template_id',
        # previously related='product_id.product_tmpl_id'
        # not anymore since the field must be considered editable for product
        configurator logic
        # without modifying the related product_id when updated.
        domain=[('sale_ok', '=', True)])
    product_uom_category_id = fields.Many2one(related='product_id.uom_id.category_id',
     depends=['product_id'])

    product_custom_attribute_value_ids = fields.One2many(
        comodel_name='product.attribute.custom.value', inverse_name=
        'sale_order_line_id',
        string="Custom Values",
        compute='_compute_custom_attribute_values',
        store=True, readonly=False, precompute=True, copy=True)
    # M2M holding the values of product.attribute with create_variant field set to
    'no_variant'
    # It allows keeping track of the extra_price associated to those attribute values
    and add them to the SO line description
    product_no_variant_attribute_value_ids = fields.Many2many(
        comodel_name='product.template.attribute.value',
        string="Extra Values",
        compute='_compute_no_variant_attribute_values',
        store=True, readonly=False, precompute=True, ondelete='restrict')

    name = fields.Text(
        string="Description",
        compute='_compute_name',
        store=True, readonly=False, required=True, precompute=True)

    product_uom_qty = fields.Float(
        string="Quantity",
        compute='_compute_product_uom_qty',
        digits='Product Unit of Measure', default=1.0,
        store=True, readonly=False, required=True, precompute=True)
    product_uom = fields.Many2one(
        comodel_name='uom.uom',
        string="Unit of Measure",
        compute='_compute_product_uom',
        store=True, readonly=False, precompute=True, ondelete='restrict',
        domain="[('category_id', '=', product_uom_category_id)]")

    # Pricing fields
    tax_id = fields.Many2many(
        comodel_name='account.tax',
        string="Taxes",
        compute='_compute_tax_id',
        store=True, readonly=False, precompute=True,
        context={'active_test': False},
        check_company=True)

    # Tech field caching pricelist rule used for price & discount computation
    pricelist_item_id = fields.Many2one(
        comodel_name='product.pricelist.item',
        compute='_compute_pricelist_item_id')

    price_unit = fields.Float(
        string="Unit Price",
        compute='_compute_price_unit',
        digits='Product Price',
        store=True, readonly=False, required=True, precompute=True)

    discount = fields.Float(
        string="Discount (%)",
        compute='_compute_discount',
        digits='Discount',
        store=True, readonly=False, precompute=True)

    price_subtotal = fields.Monetary(
        string="Subtotal",
        compute='_compute_amount',
        store=True, precompute=True)
    price_tax = fields.Float(
        string="Total Tax",
        compute='_compute_amount',
        store=True, precompute=True)
    price_total = fields.Monetary(
        string="Total",
        compute='_compute_amount',
        store=True, precompute=True)
    price_reduce_taxexcl = fields.Monetary(
        string="Price Reduce Tax excl",
        compute='_compute_price_reduce_taxexcl',
        store=True, precompute=True)
    price_reduce_taxinc = fields.Monetary(
        string="Price Reduce Tax incl",
        compute='_compute_price_reduce_taxinc',
        store=True, precompute=True)

    # Logistics/Delivery fields
    product_packaging_id = fields.Many2one(
        comodel_name='product.packaging',
        string="Packaging",
        compute='_compute_product_packaging_id',
        store=True, readonly=False, precompute=True,
        domain="[('sales', '=', True), ('product_id','=',product_id)]",
        check_company=True)
    product_packaging_qty = fields.Float(
        string="Packaging Quantity",
        compute='_compute_product_packaging_qty',
        store=True, readonly=False, precompute=True)

    customer_lead = fields.Float(
        string="Lead Time",
        compute='_compute_customer_lead',
        store=True, readonly=False, required=True, precompute=True,
        help="Number of days between the order confirmation and the shipping of the
        products to the customer")

    qty_delivered_method = fields.Selection(
        selection=[
            ('manual', "Manual"),
            ('analytic', "Analytic From Expenses"),
        ],
        string="Method to update delivered qty",
        compute='_compute_qty_delivered_method',
        store=True, precompute=True,
        help="According to product configuration, the delivered quantity can be
        automatically computed by mechanism:\n"
             "  - Manual: the quantity is set manually on the line\n"
             "  - Analytic From expenses: the quantity is the quantity sum from
             posted expenses\n"
             "  - Timesheet: the quantity is the sum of hours recorded on tasks
             linked to this sale line\n"
             "  - Stock Moves: the quantity comes from confirmed pickings\n")
```

```python
    qty_delivered = fields.Float(
        string="Delivery Quantity",
        compute='_compute_qty_delivered',
        digits='Product Unit of Measure',
        store=True, readonly=False, copy=False)

    # Analytic & Invoicing fields
    qty_invoiced = fields.Float(
        string="Invoiced Quantity",
        compute='_compute_qty_invoiced',
        digits='Product Unit of Measure',
        store=True)
    qty_to_invoice = fields.Float(
        string="Quantity To Invoice",
        compute='_compute_qty_to_invoice',
        digits='Product Unit of Measure',
        store=True)

    analytic_line_ids = fields.One2many(
        comodel_name='account.analytic.line', inverse_name='so_line',
        string="Analytic lines")

    invoice_lines = fields.Many2many(
        comodel_name='account.move.line',
        relation='sale_order_line_invoice_rel', column1='order_line_id', column2=
        'invoice_line_id',
        string="Invoice Lines",
        copy=False)
    invoice_status = fields.Selection(
        selection=[
            ('upselling', "Upselling Opportunity"),
            ('invoiced', "Fully Invoiced"),
            ('to invoice', "To Invoice"),
            ('no', "Nothing to Invoice"),
        ],
        string="Invoice Status",
        compute='_compute_invoice_status',
        store=True)

    untaxed_amount_invoiced = fields.Monetary(
        string="Untaxed Invoiced Amount",
        compute='_compute_untaxed_amount_invoiced',
        store=True)
    untaxed_amount_to_invoice = fields.Monetary(
        string="Untaxed Amount To Invoice",
        compute='_compute_untaxed_amount_to_invoice',
        store=True)

    # Technical computed fields for UX purposes (hide/make fields readonly, ...)
    product_type = fields.Selection(related='product_id.detailed_type', depends=[
    'product_id'])
    product_updatable = fields.Boolean(
        string="Can Edit Product",
        compute='_compute_product_updatable')
    product_uom_readonly = fields.Boolean(
        compute='_compute_product_uom_readonly')
    tax_calculation_rounding_method = fields.Selection(
        related='company_id.tax_calculation_rounding_method',
        string='Tax calculation rounding method', readonly=True)

    #=== COMPUTE METHODS ===#

    @api.depends('order_partner_id', 'order_id', 'product_id')
    def _compute_display_name(self):
        name_per_id = self._additional_name_per_id()
        for so_line in self.sudo():
            name = '{} - {}'.format(so_line.order_id.name, so_line.name and so_line.
            name.split('\n')[0] or so_line.product_id.name)
            additional_name = name_per_id.get(so_line.id)
            if additional_name:
                name = f'{name} {additional_name}'
            so_line.display_name = name

    @api.depends('product_id')
    def _compute_product_template_id(self):
        for line in self:
            line.product_template_id = line.product_id.product_tmpl_id

    def _search_product_template_id(self, operator, value):
        return [('product_id.product_tmpl_id', operator, value)]

    @api.depends('product_id')
    def _compute_custom_attribute_values(self):
        for line in self:
            if not line.product_id:
                line.product_custom_attribute_value_ids = False
                continue
            if not line.product_custom_attribute_value_ids:
                continue
            valid_values = line.product_id.product_tmpl_id.valid_product_template_attribute_line_ids.product_template_value_ids
            # remove the is_custom values that don't belong to this template
            for pacv in line.product_custom_attribute_value_ids:
                if pacv.custom_product_template_attribute_value_id not in valid_values:
                    line.product_custom_attribute_value_ids -= pacv

    @api.depends('product_id')
    def _compute_no_variant_attribute_values(self):
        for line in self:
            if not line.product_id:
                line.product_no_variant_attribute_value_ids = False
                continue
            if not line.product_no_variant_attribute_value_ids:
                continue
            valid_values = line.product_id.product_tmpl_id.valid_product_template_attribute_line_ids.product_template_value_ids
            # remove the no_variant attributes that don't belong to this template
            for ptav in line.product_no_variant_attribute_value_ids:
                if ptav._origin not in valid_values:
                    line.product_no_variant_attribute_value_ids -= ptav

    @api.depends('product_id')
    def _compute_name(self):
        for line in self:
            if not line.product_id:
                continue
            if not line.order_partner_id.is_public:
                line = line.with_context(lang=line.order_partner_id.lang)
            name = line._get_sale_order_line_multiline_description_sale()
            if line.is_downpayment and not line.display_type:
                context = {'lang': line.order_partner_id.lang}
                dp_state = line._get_downpayment_state()
                if dp_state == 'draft':
                    name = _("%(line_description)s (Draft)", line_description=name)
                elif dp_state == 'cancel':
                    name = _("%(line_description)s (Canceled)", line_description=name)
                else:
                    invoice = line._get_invoice_lines().move_id
                    if len(invoice) == 1 and invoice.payment_reference and invoice.invoice_date:
                        name = _(
                            "%(line_description)s (ref: %(reference)s on %(date)s)",
                            line_description=name,
                            reference=invoice.payment_reference,
                            date=format_date(line.env, invoice.invoice_date),
                        )
                del context
            line.name = name

    def _get_sale_order_line_multiline_description_sale(self):
        """ Compute a default multiline description for this sales order line.

        In most cases the product description is enough but sometimes we need to
        append information that only
        exists on the sale order line itself.
        e.g:
        - custom attributes and attributes that don't create variants, both introduced by the "product configurator"
        - in event_sale we need to know specifically the sales order line as well as the product to generate the name:
          the product is not sufficient because we also need to know the event_id and the event_ticket_id (both which belong to the sale order line).
        """
        self.ensure_one()
        return self.product_id.get_product_multiline_description_sale() + self._get_sale_order_line_multiline_description_variants()

    def _get_sale_order_line_multiline_description_variants(self):
        """When using no_variant attributes or is_custom values, the product
        itself is not sufficient to create the description: we need to add
        information about those special attributes and values.

        :return: the description related to special variant attributes/values
        :rtype: string
        """
        if not self.product_custom_attribute_value_ids and not self.product_no_variant_attribute_value_ids:
            return ""

        name = "\n"

        custom_ptavs = self.product_custom_attribute_value_ids.custom_product_template_attribute_value_id
        no_variant_ptavs = self.product_no_variant_attribute_value_ids._origin
        multi_ptavs = no_variant_ptavs.filtered(lambda ptav: ptav.display_type == 'multi').sorted()

        # display the no_variant attributes, except those that are also
        # displayed by a custom (avoid duplicate description)
        for ptav in (no_variant_ptavs - multi_ptavs - custom_ptavs):
            name += "\n" + ptav.display_name

        # display the selected values per attribute on a single for a multi checkbox
        for pta, ptavs in groupby(multi_ptavs, lambda ptav: ptav.attribute_id):
            name += "\n" + _(
                "%(attribute)s: %(values)s",
                attribute=pta.name,
                values=", ".join(ptav.name for ptav in ptavs)
            )

        # Sort the values according to _order settings, because it doesn't work for virtual records in onchange
        sorted_custom_ptav = self.product_custom_attribute_value_ids.custom_product_template_attribute_value_id.sorted()
        for patv in sorted_custom_ptav:
            pacv = self.product_custom_attribute_value_ids.filtered(lambda pcav: pcav.custom_product_template_attribute_value_id == patv)
            name += "\n" + pacv.display_name

        return name

    @api.depends('display_type', 'product_id', 'product_packaging_qty')
    def _compute_product_uom_qty(self):
        for line in self:
            if line.display_type:
                line.product_uom_qty = 0.0
                continue

            if not line.product_packaging_id:
                continue
            packaging_uom = line.product_packaging_id.product_uom_id
            qty_per_packaging = line.product_packaging_id.qty
            product_uom_qty = packaging_uom._compute_quantity(
                line.product_packaging_qty * qty_per_packaging, line.product_uom)
            if float_compare(product_uom_qty, line.product_uom_qty, precision_rounding
            =line.product_uom.rounding) != 0:
                line.product_uom_qty = product_uom_qty

    @api.depends('product_id')
    def _compute_product_uom(self):
        for line in self:
            if not line.product_uom or (line.product_id.uom_id.id != line.product_uom.id):
                line.product_uom = line.product_id.uom_id

    @api.depends('product_id', 'company_id')
    def _compute_tax_id(self):
        taxes_by_product_company = defaultdict(lambda: self.env['account.tax'])
        lines_by_company = defaultdict(lambda: self.env['sale.order.line'])
        cached_taxes = {}
        for line in self:
            lines_by_company[line.company_id] += line
        for product in self.product_id:
            for tax in product.taxes_id:
                taxes_by_product_company[(product, tax.company_id)] += tax
        for company, lines in lines_by_company.items():
            for line in lines.with_company(company):
                taxes, comp = None, company
                while not taxes and comp:
                    taxes = taxes_by_product_company[(line.product_id, comp)]
                    comp = comp.parent_id
                if not line.product_id or not taxes:
                    # Nothing to map
                    line.tax_id = False
                    continue
                fiscal_position = line.order_id.fiscal_position_id
                cache_key = (fiscal_position.id, company.id, tuple(taxes.ids))
                if cache_key in cached_taxes:
                    result = cached_taxes[cache_key]
                else:
                    result = fiscal_position.map_tax(taxes)
                    cached_taxes[cache_key] = result
                # If company_id is set, always filter taxes by the company
                line.tax_id = result

    @api.depends('product_id', 'product_uom', 'product_uom_qty')
    def _compute_pricelist_item_id(self):
        for line in self:
            if not line.product_id or line.display_type or not line.order_id.pricelist_id:
                line.pricelist_item_id = False
            else:
                line.pricelist_item_id = line.order_id.pricelist_id._get_product_rule(
                    line.product_id,
                    quantity=line.product_uom_qty or 1.0,
                    uom=line.product_uom,
                    date=line.order_id.date_order,
                )

    @api.depends('product_id', 'product_uom', 'product_uom_qty')
    def _compute_price_unit(self):
        for line in self:
            # check if there is already invoiced amount. if so, the price shouldn't change as it might have been
            # manually edited
            if line.qty_invoiced > 0:
                continue
            if not line.product_uom or not line.product_id:
                line.price_unit = 0.0
            else:
                price = line.with_company(line.company_id)._get_display_price()
                line.price_unit = line.product_id._get_tax_included_unit_price(
                    line.company_id or line.env.company,
                    line.order_id.currency_id,
                    line.order_id.date_order,
                    'sale',
                    fiscal_position=line.order_id.fiscal_position_id,
```

```python
                    product_price_unit=price,
                    product_currency=line.currency_id
                )

    def _get_display_price(self):
        """Compute the displayed unit price for a given line.

        Overridden in custom flows:
        * where the price is not specified by the pricelist
        * where the discount is not specified by the pricelist

        Note: self.ensure_one()
        """
        self.ensure_one()

        pricelist_price = self._get_pricelist_price()

        if self.order_id.pricelist_id.discount_policy == 'with_discount':
            return pricelist_price

        if not self.pricelist_item_id:
            # No pricelist rule found => no discount from pricelist
            return pricelist_price

        base_price = self._get_pricelist_price_before_discount()

        # negative discounts (= surcharge) are included in the display price
        return max(base_price, pricelist_price)

    def _get_pricelist_price(self):
        """Compute the price given by the pricelist for the given line information.

        :return: the product sales price in the order currency (without taxes)
        :rtype: float
        """
        self.ensure_one()
        self.product_id.ensure_one()

        price = self.pricelist_item_id._compute_price(
            product=self.product_id.with_context(**self._get_product_price_context()),
            quantity=self.product_uom_qty or 1.0,
            uom=self.product_uom,
            date=self.order_id.date_order,
            currency=self.currency_id,
        )

        return price

    def _get_product_price_context(self):
        """Gives the context for product price computation.

        :return: additional context to consider extra prices from attributes in the base product price.
        :rtype: dict
        """
        self.ensure_one()
        return self.product_id._get_product_price_context(
            self.product_no_variant_attribute_value_ids,
        )

    def _get_pricelist_price_context(self):
        """DO NOT USE in new code, this contextual logic should be dropped or heavily refactored soon"""
        self.ensure_one()
        return {
            'pricelist': self.order_id.pricelist_id.id,
            'uom': self.product_uom.id,
            'quantity': self.product_uom_qty,
            'date': self.order_id.date_order,
        }

    def _get_pricelist_price_before_discount(self):
        """Compute the price used as base for the pricelist price computation.

        :return: the product sales price in the order currency (without taxes)
        :rtype: float
        """
        self.ensure_one()
        self.product_id.ensure_one()

        return self.pricelist_item_id._compute_price_before_discount(
            product=self.product_id.with_context(**self._get_product_price_context()),
            quantity=self.product_uom_qty or 1.0,
            uom=self.product_uom,
            date=self.order_id.date_order,
            currency=self.currency_id,
        )

    @api.depends('product_id', 'product_uom', 'product_uom_qty')
    def _compute_discount(self):
        for line in self:
            if not line.product_id or line.display_type:
                line.discount = 0.0

            if not (
                line.order_id.pricelist_id
                and line.order_id.pricelist_id.discount_policy == 'without_discount'
            ):
                continue

            line.discount = 0.0

            if not line.pricelist_item_id:
                # No pricelist rule was found for the product
                # therefore, the pricelist didn't apply any discount/change
                # to the existing sales price.
                continue

            line = line.with_company(line.company_id)
            pricelist_price = line._get_pricelist_price()
            base_price = line._get_pricelist_price_before_discount()

            if base_price != 0:  # Avoid division by zero
                discount = (base_price - pricelist_price) / base_price * 100
                if (discount > 0 and base_price > 0) or (discount < 0 and base_price < 0):
                    # only show negative discounts if price is negative
                    # otherwise it's a surcharge which shouldn't be shown to the customer
                    line.discount = discount

    def _convert_to_tax_base_line_dict(self, **kwargs):
        """ Convert the current record to a dictionary in order to use the generic taxes computation method
        defined on account.tax.

        :return: A python dictionary.
        """
        self.ensure_one()
        return self.env['account.tax']._convert_to_tax_base_line_dict(
            self,
            partner=self.order_id.partner_id,
            currency=self.order_id.currency_id,
            product=self.product_id,
            taxes=self.tax_id,
            price_unit=self.price_unit,
            quantity=self.product_uom_qty,
            discount=self.discount,
            price_subtotal=self.price_subtotal,
            **kwargs,
        )

    @api.depends('product_uom_qty', 'discount', 'price_unit', 'tax_id')
    def _compute_amount(self):
        """
        Compute the amounts of the SO line.
        """
        for line in self:
            tax_results = self.env['account.tax']._compute_taxes([
                line._convert_to_tax_base_line_dict()
            ])
            totals = list(tax_results['totals'].values())[0]
            amount_untaxed = totals['amount_untaxed']
            amount_tax = totals['amount_tax']

            line.update({
                'price_subtotal': amount_untaxed,
                'price_tax': amount_tax,
                'price_total': amount_untaxed + amount_tax,
            })

    @api.depends('price_subtotal', 'product_uom_qty')
    def _compute_price_reduce_taxexcl(self):
        for line in self:
            line.price_reduce_taxexcl = line.price_subtotal / line.product_uom_qty if line.product_uom_qty else 0.0

    @api.depends('price_total', 'product_uom_qty')
    def _compute_price_reduce_taxinc(self):
        for line in self:
            line.price_reduce_taxinc = line.price_total / line.product_uom_qty if line.product_uom_qty else 0.0

    @api.depends('product_id', 'product_uom_qty', 'product_uom')
    def _compute_product_packaging_id(self):
        for line in self:
            # remove packaging if not match the product
            if line.product_packaging_id.product_id != line.product_id:
                line.product_packaging_id = False
            # suggest biggest suitable packaging matching the SO's company
            if line.product_id and line.product_uom_qty and line.product_uom:
                suggested_packaging = line.product_id.packaging_ids\
                        .filtered(lambda p: p.sales and (p.product_id.company_id <= p.company_id <= line.company_id))\
                        ._find_suitable_product_packaging(line.product_uom_qty, line.product_uom)
                line.product_packaging_id = suggested_packaging or line.product_packaging_id

    @api.depends('product_packaging_id', 'product_uom', 'product_uom_qty')
    def _compute_product_packaging_qty(self):
        self.product_packaging_qty = 0
        for line in self:
            if not line.product_packaging_id:
                continue
            line.product_packaging_qty = line.product_packaging_id._compute_qty(line.product_uom_qty, line.product_uom)

    # This computed default is necessary to have a clean computation inheritance
    # (cf sale_stock) instead of simply removing the default and specifying
    # the compute attribute & method in sale_stock.
    def _compute_customer_lead(self):
        self.customer_lead = 0.0

    @api.depends('is_expense')
    def _compute_qty_delivered_method(self):
        """ Sale module compute delivered qty for product [('type', 'in', ['consu']), ('service_type', '=', 'manual')]
                - consu + expense_policy : analytic (sum of analytic unit_amount)
                - consu + no expense_policy : manual (set manually on SOL)
                - service (+ service_type='manual', the only available option) : manual

            This is true when only sale is installed: sale_stock redifine the behavior for 'consu' type,
            and sale_timesheet implements the behavior of 'service' +
            service_type=timesheet.
        """
        for line in self:
            if line.is_expense:
                line.qty_delivered_method = 'analytic'
            else:  # service and consu
                line.qty_delivered_method = 'manual'

    @api.depends(
        'qty_delivered_method',
        'analytic_line_ids.so_line',
        'analytic_line_ids.unit_amount',
        'analytic_line_ids.product_uom_id')
    def _compute_qty_delivered(self):
        """ This method compute the delivered quantity of the SO lines: it covers the case provide by sale module, aka
            expense/vendor bills (sum of unit_amount of AAL), and manual case.
            This method should be overridden to provide other way to automatically compute delivered qty. Overrides should
            take their concerned so lines, compute and set the `qty_delivered` field, and call super with the remaining
            records.
        """
        # compute for analytic lines
        lines_by_analytic = self.filtered(lambda sol: sol.qty_delivered_method == 'analytic')
        mapping = lines_by_analytic._get_delivered_quantity_by_analytic([('amount', '<=', 0.0)])
        for so_line in lines_by_analytic:
            so_line.qty_delivered = mapping.get(so_line.id or so_line._origin.id, 0.0)

    def _get_downpayment_state(self):
        self.ensure_one()

        if self.display_type:
            return ''

        invoice_lines = self._get_invoice_lines()
        if all(line.parent_state == 'draft' for line in invoice_lines):
            return 'draft'
        if all(line.parent_state == 'cancel' for line in invoice_lines):
            return 'cancel'

        return ''

    def _get_delivered_quantity_by_analytic(self, additional_domain):
        """ Compute and write the delivered quantity of current SO lines, based on their related
            analytic lines.
            :param additional_domain: domain to restrict AAL to include in computation (required since timesheet is an AAL with a project ...)
        """
        result = defaultdict(float)

        # avoid recomputation if no SO lines concerned
        if not self:
            return result

        # group analytic lines by product uom and so line
        domain = expression.AND([[('so_line', 'in', self.ids)], additional_domain])
        data = self.env['account.analytic.line']._read_group(
            domain,
            ['product_uom_id', 'so_line'],
            ['unit_amount:sum', 'move_line_id:count_distinct', '__count'],
        )

        # convert uom and sum all unit_amount of analytic lines to get the delivered qty of SO lines
        for uom, so_line, unit_amount_sum, move_line_id_count_distinct, count in data:
            if not uom:
                continue
            # avoid counting unit_amount twice when dealing with multiple analytic
```

```python
            lines on the same move line
            if move_line_id_count_distinct == 1 and count > 1:
                qty = unit_amount_sum / count
            else:
                qty = unit_amount_sum
            if so_line.product_uom.category_id == uom.category_id:
                qty = uom._compute_quantity(qty, so_line.product_uom, rounding_method='HALF-UP')
            result[so_line.id] += qty

        return result

    @api.depends('invoice_lines.move_id.state', 'invoice_lines.quantity')
    def _compute_qty_invoiced(self):
        """
        Compute the quantity invoiced. If case of a refund, the quantity invoiced is decreased. Note
        that this is the case only if the refund is generated from the SO and that is intentional: if
        a refund made would automatically decrease the invoiced quantity, then there is a risk of reinvoicing
        it automatically, which may not be wanted at all. That's why the refund has to be created from the SO
        """
        for line in self:
            qty_invoiced = 0.0
            for invoice_line in line._get_invoice_lines():
                if invoice_line.move_id.state != 'cancel' or invoice_line.move_id.payment_state == 'invoicing_legacy':
                    if invoice_line.move_id.move_type == 'out_invoice':
                        qty_invoiced += invoice_line.product_uom_id._compute_quantity(invoice_line.quantity, line.product_uom)
                    elif invoice_line.move_id.move_type == 'out_refund':
                        qty_invoiced -= invoice_line.product_uom_id._compute_quantity(invoice_line.quantity, line.product_uom)
            line.qty_invoiced = qty_invoiced

    def _get_invoice_lines(self):
        self.ensure_one()
        if self._context.get('accrual_entry_date'):
            return self.invoice_lines.filtered(
                lambda l: l.move_id.invoice_date and l.move_id.invoice_date <= self._context['accrual_entry_date']
            )
        else:
            return self.invoice_lines

    # no trigger product_id.invoice_policy to avoid retroactively changing SO
    @api.depends('qty_invoiced', 'qty_delivered', 'product_uom_qty', 'state')
    def _compute_qty_to_invoice(self):
        """
        Compute the quantity to invoice. If the invoice policy is order, the quantity to invoice is
        calculated from the ordered quantity. Otherwise, the quantity delivered is used.
        """
        for line in self:
            if line.state == 'sale' and not line.display_type:
                if line.product_id.invoice_policy == 'order':
                    line.qty_to_invoice = line.product_uom_qty - line.qty_invoiced
                else:
                    line.qty_to_invoice = line.qty_delivered - line.qty_invoiced
            else:
                line.qty_to_invoice = 0

    @api.depends('state', 'product_uom_qty', 'qty_delivered', 'qty_to_invoice', 'qty_invoiced')
    def _compute_invoice_status(self):
        """
        Compute the invoice status of a SO line. Possible statuses:
        - no: if the SO is not in status 'sale', we consider that there is nothing to
          invoice. This is also the default value if the conditions of no other
          status is met.
        - to invoice: we refer to the quantity to invoice of the line. Refer to method
          `_compute_qty_to_invoice()` for more information on how this quantity is calculated.
        - upselling: this is possible only for a product invoiced on ordered quantities for which
          we delivered more than expected. The could arise if, for example, a project took more
          time than expected but we decided not to invoice the extra cost to the client. This
          occurs only in state 'sale', the upselling opportunity is removed from the list.
        - invoiced: the quantity invoiced is larger or equal to the quantity ordered.
        """
        precision = self.env['decimal.precision'].precision_get('Product Unit of Measure')
        for line in self:
            if line.state != 'sale':
                line.invoice_status = 'no'
            elif line.is_downpayment and line.untaxed_amount_to_invoice == 0:
                line.invoice_status = 'invoiced'
            elif not float_is_zero(line.qty_to_invoice, precision_digits=precision):
                line.invoice_status = 'to invoice'
            elif line.state == 'sale' and line.product_id.invoice_policy == 'order' and\
                    line.product_uom_qty >= 0.0 and\
                    float_compare(line.qty_delivered, line.product_uom_qty, precision_digits=precision) == 1:
                line.invoice_status = 'upselling'
            elif float_compare(line.qty_invoiced, line.product_uom_qty, precision_digits=precision) >= 0:
                line.invoice_status = 'invoiced'
            else:
                line.invoice_status = 'no'

    @api.depends('invoice_lines', 'invoice_lines.price_total', 'invoice_lines.move_id.state', 'invoice_lines.move_id.move_type')
    def _compute_untaxed_amount_invoiced(self):
        """ Compute the untaxed amount already invoiced from the sale order line, taking the refund attached
            the so line into account. This amount is computed as
                SUM(inv_line.price_subtotal) - SUM(ref_line.price_subtotal)
            where
                `inv_line` is a customer invoice line linked to the SO line
                `ref_line` is a customer credit note (refund) line linked to the SO line
        """
        for line in self:
            amount_invoiced = 0.0
            for invoice_line in line._get_invoice_lines():
                if invoice_line.move_id.state == 'posted':
                    invoice_date = invoice_line.move_id.invoice_date or fields.Date.today()
                    if invoice_line.move_id.move_type == 'out_invoice':
                        amount_invoiced += invoice_line.currency_id._convert(invoice_line.price_subtotal, line.currency_id, line.company_id, invoice_date)
                    elif invoice_line.move_id.move_type == 'out_refund':
                        amount_invoiced -= invoice_line.currency_id._convert(invoice_line.price_subtotal, line.currency_id, line.company_id, invoice_date)
            line.untaxed_amount_invoiced = amount_invoiced

    @api.depends('state', 'product_id', 'untaxed_amount_invoiced', 'qty_delivered', 'product_uom_qty', 'price_unit')
    def _compute_untaxed_amount_to_invoice(self):
        """ Total of remaining amount to invoice on the sale order line (taxes excl.) as
                total_sol - amount already invoiced
            where Total_sol depends on the invoice policy of the product.

            Note: Draft invoice are ignored on purpose, the 'to invoice' amount should
```

```python
            come only from the SO lines.
        """
        for line in self:
            amount_to_invoice = 0.0
            if line.state == 'sale':
                # Note: do not use price_subtotal field as it returns zero when the ordered quantity is
                # zero. It causes problem for expense line (e.i.: ordered qty = 0, deli qty = 4,
                # price_unit = 20 ; subtotal is zero), but when you can invoice the line, you see an
                # amount and not zero. Since we compute untaxed amount, we can use directly the price
                # reduce (to include discount) without using `compute_all()` method on taxes.
                price_subtotal = 0.0
                uom_qty_to_consider = line.qty_delivered if line.product_id.invoice_policy == 'delivery' else line.product_uom_qty
                price_reduce = line.price_unit * (1 - (line.discount or 0.0) / 100.0)
                price_subtotal = price_reduce * uom_qty_to_consider
                if len(line.tax_id.filtered(lambda tax: tax.price_include)) > 0:
                    # As included taxes are not excluded from the computed subtotal, `compute_all()` method
                    # has to be called to retrieve the subtotal without them.
                    # `price_reduce_taxexcl` cannot be used as it is computed from `price_subtotal` field. (see upper Note)
                    price_subtotal = line.tax_id.compute_all(
                        price_reduce,
                        currency=line.currency_id,
                        quantity=uom_qty_to_consider,
                        product=line.product_id,
                        partner=line.order_id.partner_shipping_id)['total_excluded']
                inv_lines = line._get_invoice_lines()
                if any(inv_lines.mapped(lambda l: l.discount != line.discount)):
                    # In case of re-invoicing with different discount we try to calculate manually the
                    # remaining amount to invoice
                    amount = 0
                    for l in inv_lines:
                        if len(l.tax_ids.filtered(lambda tax: tax.price_include)) > 0:
                            amount += l.tax_ids.compute_all(l.currency_id._convert(l.price_unit, line.currency_id, line.company_id, l.date or fields.Date.today(), round=False) * l.quantity)['total_excluded']
                        else:
                            amount += l.currency_id._convert(l.price_unit, line.currency_id, line.company_id, l.date or fields.Date.today(), round=False) * l.quantity

                    amount_to_invoice = max(price_subtotal - amount, 0)
                else:
                    amount_to_invoice = price_subtotal - line.untaxed_amount_invoiced

            line.untaxed_amount_to_invoice = amount_to_invoice

    @api.depends('order_id.partner_id', 'product_id')
    def _compute_analytic_distribution(self):
        for line in self:
            if not line.display_type:
                distribution = line.env['account.analytic.distribution.model']._get_distribution({
                    "product_id": line.product_id.id,
                    "product_categ_id": line.product_id.categ_id.id,
                    "partner_id": line.order_id.partner_id.id,
                    "partner_category_id": line.order_id.partner_id.category_id.ids,
                    "company_id": line.company_id.id,
                })
                line.analytic_distribution = distribution or line.analytic_distribution

    @api.depends('product_id', 'state', 'qty_invoiced', 'qty_delivered')
    def _compute_product_updatable(self):
        for line in self:
            if line.state == 'cancel':
                line.product_updatable = False
            elif line.state == 'sale' and (
                line.order_id.locked
                or line.qty_invoiced > 0
                or line.qty_delivered > 0
            ):
                line.product_updatable = False
            else:
                line.product_updatable = True

    @api.depends('state')
    def _compute_product_uom_readonly(self):
        for line in self:
            # line.ids checks whether it's a new record not yet saved
            line.product_uom_readonly = line.ids and line.state in ['sale', 'cancel']

    #=== CONSTRAINT METHODS ===#

    #=== ONCHANGE METHODS ===#

    @api.onchange('product_id')
    def _onchange_product_id_warning(self):
        if not self.product_id:
            return

        product = self.product_id
        if product.sale_line_warn != 'no-message':
            if product.sale_line_warn == 'block':
                self.product_id = False

            return {
                'warning': {
                    'title': _("Warning for %s", product.name),
                    'message': product.sale_line_warn_msg,
                }
            }

    @api.onchange('product_packaging_id')
    def _onchange_product_packaging_id(self):
        if self.product_packaging_id and self.product_uom_qty:
            newqty = self.product_packaging_id._check_qty(self.product_uom_qty, self.
            product_uom, "UP")
            if float_compare(newqty, self.product_uom_qty, precision_rounding=self.
            product_uom.rounding) != 0:
                return {
                    'warning': {
                        'title': _('Warning'),
                        'message': _(
                            "This product is packaged by %(pack_size).2f
                            %(pack_name)s. You should sell %(quantity).2f %(unit)s.",
                            pack_size=self.product_packaging_id.qty,
                            pack_name=self.product_id.uom_id.name,
                            quantity=newqty,
                            unit=self.product_uom.name
                        ),
                    },
                }

    #=== CRUD METHODS ===#

    @api.model_create_multi
    def create(self, vals_list):
        for vals in vals_list:
            if vals.get('display_type') or self.default_get(['display_type']).get(
            'display_type'):
                vals['product_uom_qty'] = 0.0

        lines = super().create(vals_list)
        if self.env.context.get('sale_no_log_for_new_lines'):
            return lines
```

```python

        for line in lines:
            if line.product_id and line.state == 'sale':
                msg = _("Extra line with %s", line.product_id.display_name)
                line.order_id.message_post(body=msg)
                # create an analytic account if at least an expense product
                if line.product_id.expense_policy not in [False, 'no'] and not line.order_id.analytic_account_id:
                    line.order_id._create_analytic_account()

        return lines

    def write(self, values):
        if 'display_type' in values and self.filtered(lambda line: line.display_type != values.get('display_type')):
            raise UserError(_("You cannot change the type of a sale order line. Instead you should delete the current line and create a new line of the proper type."))

        if 'product_uom_qty' in values:
            precision = self.env['decimal.precision'].precision_get('Product Unit of Measure')
            self.filtered(
                lambda r: r.state == 'sale' and float_compare(r.product_uom_qty, values['product_uom_qty'], precision_digits=precision) != 0)._update_line_quantity(values)

        # Prevent writing on a locked SO.
        protected_fields = self._get_protected_fields()
        if any(self.order_id.mapped('locked')) and any(f in values.keys() for f in protected_fields):
            protected_fields_modified = list(set(protected_fields) & set(values.keys()))

            if 'name' in protected_fields_modified and all(self.mapped('is_downpayment')):
                protected_fields_modified.remove('name')

            fields = self.env['ir.model.fields'].sudo().search([
                ('name', 'in', protected_fields_modified), ('model', '=', self._name)
            ])
            if fields:
                raise UserError(
                    _('It is forbidden to modify the following fields in a locked order:\n%s',
                      '\n'.join(fields.mapped('field_description')))
                )

        result = super().write(values)

        # Don't recompute the package_id if we are setting the quantity of the items and the quantity of packages
        if 'product_uom_qty' in values and 'product_packaging_qty' in values and 'product_packaging_id' not in values:
            self.env.remove_to_compute(self._fields['product_packaging_id'], self)

        return result

    def _get_protected_fields(self):
        """ Give the fields that should not be modified on a locked SO.

        :returns: list of field names
        :rtype: list
        """
        return [
            'product_id', 'name', 'price_unit', 'product_uom', 'product_uom_qty',
            'tax_id', 'analytic_distribution'
        ]

    def _update_line_quantity(self, values):
        orders = self.mapped('order_id')
        for order in orders:
            order_lines = self.filtered(lambda x: x.order_id == order)
            msg = Markup("<b>%s</b><ul>") % _("The ordered quantity has been updated.")
            for line in order_lines:
                msg += Markup("<li> %s: <br/>") % line.product_id.display_name
                msg += _(
                    "Ordered Quantity: %(old_qty)s -> %(new_qty)s",
                    old_qty=line.product_uom_qty,
                    new_qty=values["product_uom_qty"]
                ) + Markup("<br/>")
                if line.product_id.type in ('consu', 'product'):
                    msg += _("Delivered Quantity: %s", line.qty_delivered) + Markup("<br/>")
                msg += _("Invoiced Quantity: %s", line.qty_invoiced) + Markup("<br/>")
            msg += Markup("</ul>")
            order.message_post(body=msg)

    def _check_line_unlink(self):
        """ Check whether given lines can be deleted or not.

        * Lines cannot be deleted if the order is confirmed.
        * Down payment lines who have not yet been invoiced bypass that exception.
        * Sections and Notes can always be deleted.

        :returns: Sales Order Lines that cannot be deleted
        :rtype: `sale.order.line` recordset
        """
        return self.filtered(
            lambda line:
                line.state == 'sale'
                and (line.invoice_lines or not line.is_downpayment)
                and not line.display_type
        )

    @api.ondelete(at_uninstall=False)
    def _unlink_except_confirmed(self):
        if self._check_line_unlink():
            raise UserError(_("Once a sales order is confirmed, you can't remove one of its lines (we need to track if something gets invoiced or delivered).\n\
                Set the quantity to 0 instead."))

    #=== ACTION METHODS ===#

    def action_add_from_catalog(self):
        order = self.env['sale.order'].browse(self.env.context.get('order_id'))
        return order.action_add_from_catalog()

    #=== BUSINESS METHODS ===#

    def _expected_date(self):
        self.ensure_one()
        if self.state == 'sale' and self.order_id.date_order:
            order_date = self.order_id.date_order
        else:
            order_date = fields.Datetime.now()
        return order_date + timedelta(days=self.customer_lead or 0.0)

    def compute_uom_qty(self, new_qty, stock_move, rounding=True):
        return self.product_uom._compute_quantity(new_qty, stock_move.product_uom, rounding)

    def _get_invoice_line_sequence(self, new=0, old=0):
        """
        Method intended to be overridden in third-party module if we want to prevent the resequencing
        of invoice lines.

        :param int new:   the new line sequence
        :param int old:   the old line sequence

        :return:          the sequence of the SO line, by default the new one.
        """
        return new or old

    def _prepare_invoice_line(self, **optional_values):
        """Prepare the values to create the new invoice line for a sales order line.

        :param optional_values: any parameter that should be added to the returned
        invoice line
        :rtype: dict
        """
        self.ensure_one()
        res = {
            'display_type': self.display_type or 'product',
            'sequence': self.sequence,
            'name': self.name,
            'product_id': self.product_id.id,
            'product_uom_id': self.product_uom.id,
            'quantity': self.qty_to_invoice,
            'discount': self.discount,
            'price_unit': self.price_unit,
            'tax_ids': [Command.set(self.tax_id.ids)],
            'sale_line_ids': [Command.link(self.id)],
            'is_downpayment': self.is_downpayment,
        }
        analytic_account_id = self.order_id.analytic_account_id.id
        if self.analytic_distribution and not self.display_type:
            res['analytic_distribution'] = self.analytic_distribution
        if analytic_account_id and not self.display_type:
            analytic_account_id = str(analytic_account_id)
            if 'analytic_distribution' in res:
                res['analytic_distribution'][analytic_account_id] = res[
                'analytic_distribution'].get(analytic_account_id, 0) + 100
            else:
                res['analytic_distribution'] = {analytic_account_id: 100}
        if optional_values:
            res.update(optional_values)
        if self.display_type:
            res['account_id'] = False
        return res

    def _prepare_procurement_values(self, group_id=False):
        """ Prepare specific key for moves or other components that will be created
        from a stock rule
        coming from a sale order line. This method could be override in order to add
        other custom key that could
        be used in move/po creation.
        """
        return {}

    def _validate_analytic_distribution(self):
        for line in self.filtered(lambda l: not l.display_type and l.state in ['draft'
        , 'sent']):
            line._validate_distribution(**{
                'product': line.product_id.id,
                'business_domain': 'sale_order',
                'company_id': line.company_id.id,
            })

    #=== CORE METHODS OVERRIDES ===#

    def _get_partner_display(self):
        self.ensure_one()
        commercial_partner = self.order_partner_id.commercial_partner_id
        return f'({commercial_partner.ref or commercial_partner.name})'

    def _additional_name_per_id(self):
        return {
            so_line.id: so_line._get_partner_display()
            for so_line in self
        }

    #=== HOOKS ===#

    def _is_delivery(self):
        self.ensure_one()
        return False

    def _is_not_sellable_line(self):
        # True if the line is a computed line (reward, delivery, ...) that user cannot add manually
        return False

    def _get_product_catalog_lines_data(self, **kwargs):
        """ Return information about sale order lines in `self`.

        If `self` is empty, this method returns only the default value(s) needed for the product
        catalog. In this case, the quantity that equals 0.

        Otherwise, it returns a quantity and a price based on the product of the SOL(s) and whether
        the product is read-only or not.

        A product is considered read-only if the order is considered read-only (see
        ``SaleOrder._is_readonly`` for more details) or if `self` contains multiple records.

        Note: This method cannot be called with multiple records that have different products linked.

        :raise odoo.exceptions.ValueError: ``len(self.product_id) != 1``
        :rtype: dict
        :return: A dict with the following structure:
            {
                'quantity': float,
                'price': float,
                'readOnly': bool,
            }
        """
        if len(self) == 1:
            return {
                'quantity': self.product_uom_qty,
                'price': self.price_unit,
                'readOnly': self.order_id._is_readonly(),
            }
        elif self:
            self.product_id.ensure_one()
            order_line = self[0]
            order = order_line.order_id
            return {
                'readOnly': True,
                'price': order.pricelist_id._get_product_price(
                    product=order_line.product_id,
                    quantity=1.0,
                    currency=order.currency_id,
                    date=order.date_order,
                    **kwargs,
                ),
                'quantity': sum(
                    self.mapped(
                        lambda line: line.product_uom._compute_quantity(
                            qty=line.product_uom_qty,
                            to_unit=line.product_id.uom_id,
                        )
                    )
                ),
            }
        else:
            return {
                'quantity': 0,
                # price will be computed in batch with pricelist utils so not given here
            }

    #=== TOOLING ===#

    def _convert_to_sol_currency(self, amount, currency):
        """Convert the given amount from the given currency to the SO(L) currency.

        :param float amount: the amount to convert
        :param currency: currency in which the given amount is expressed
        :type currency: `res.currency` record
        :returns: converted amount
        :rtype: float
        """
        self.ensure_one()
        to_currency = self.currency_id or self.order_id.currency_id
        if currency and to_currency and currency != to_currency:
            conversion_date = self.order_id.date_order or fields.Date.context_today(self)
            company = self.company_id or self.order_id.company_id or self.env.company
            return currency._convert(
                from_amount=amount,
                to_currency=to_currency,
                company=company,
                date=conversion_date,
                round=False,
            )
        return amount

    def has_valued_move_ids(self):
        return self.move_ids


# sale_report.py
 # -*- coding: utf-8 -*-
# Part of Odoo. See LICENSE file for full copyright and licensing details.

from odoo import api, fields, models, tools
from odoo.addons.sale.models.sale_order import SALE_ORDER_STATE


class SaleReport(models.Model):
    _name = "sale.report"
    _description = "Sales Analysis Report"
    _auto = False
    _rec_name = 'date'
    _order = 'date desc'

    @api.model
    def _get_done_states(self):
        return ['sale']

    # sale.order fields
    name = fields.Char(string="Order Reference", readonly=True)
    date = fields.Datetime(string="Order Date", readonly=True)
    partner_id = fields.Many2one(comodel_name='res.partner', string="Customer", readonly=True)
    company_id = fields.Many2one(comodel_name='res.company', readonly=True)
    pricelist_id = fields.Many2one(comodel_name='product.pricelist', readonly=True)
    team_id = fields.Many2one(comodel_name='crm.team', string="Sales Team", readonly=True)
    user_id = fields.Many2one(comodel_name='res.users', string="Salesperson", readonly=True)
    state = fields.Selection(selection=SALE_ORDER_STATE, string="Status", readonly=True)
    analytic_account_id = fields.Many2one(
        comodel_name='account.analytic.account', string="Analytic Account", readonly=True)
    invoice_status = fields.Selection(
        selection=[
            ('upselling', "Upselling Opportunity"),
            ('invoiced', "Fully Invoiced"),
            ('to invoice', "To Invoice"),
            ('no', "Nothing to Invoice"),
        ], string="Invoice Status", readonly=True)
    campaign_id = fields.Many2one(comodel_name='utm.campaign', string="Campaign",
    readonly=True)
    medium_id = fields.Many2one(comodel_name='utm.medium', string="Medium", readonly=
    True)
    source_id = fields.Many2one(comodel_name='utm.source', string="Source", readonly=
    True)

    # res.partner fields
    commercial_partner_id = fields.Many2one(
        comodel_name='res.partner', string="Customer Entity", readonly=True)
    country_id = fields.Many2one(
        comodel_name='res.country', string="Customer Country", readonly=True)
    industry_id = fields.Many2one(
        comodel_name='res.partner.industry', string="Customer Industry", readonly=True
        )
    partner_zip = fields.Char(string="Customer ZIP", readonly=True)
    state_id = fields.Many2one(comodel_name='res.country.state', string="Customer
    State", readonly=True)

    # sale.order.line fields
    order_reference = fields.Reference(string='Related Order', selection=[(
    'sale.order', 'Sales Order')], group_operator="count_distinct")

    categ_id = fields.Many2one(
        comodel_name='product.category', string="Product Category", readonly=True)
    product_id = fields.Many2one(
        comodel_name='product.product', string="Product Variant", readonly=True)
    product_tmpl_id = fields.Many2one(
        comodel_name='product.template', string="Product", readonly=True)
    product_uom = fields.Many2one(comodel_name='uom.uom', string="Unit of Measure",
    readonly=True)
    product_uom_qty = fields.Float(string="Qty Ordered", readonly=True)
    qty_to_deliver = fields.Float(string="Qty To Deliver", readonly=True)
    qty_delivered = fields.Float(string="Qty Delivered", readonly=True)
    qty_to_invoice = fields.Float(string="Qty To Invoice", readonly=True)
    qty_invoiced = fields.Float(string="Qty Invoiced", readonly=True)
    price_subtotal = fields.Monetary(string="Untaxed Total", readonly=True)
    price_total = fields.Monetary(string="Total", readonly=True)
    untaxed_amount_to_invoice = fields.Monetary(string="Untaxed Amount To Invoice",
    readonly=True)
    untaxed_amount_invoiced = fields.Monetary(string="Untaxed Amount Invoiced",
    readonly=True)

    weight = fields.Float(string="Gross Weight", readonly=True)
    volume = fields.Float(string="Volume", readonly=True)

    discount = fields.Float(string="Discount %", readonly=True)
    discount_amount = fields.Monetary(string="Discount Amount", readonly=True)

    # aggregates or computed fields
    nbr = fields.Integer(string="# of Lines", readonly=True)
    currency_id = fields.Many2one(comodel_name='res.currency', compute=
    '_compute_currency_id')

    @api.depends_context('allowed_company_ids')
    def _compute_currency_id(self):
        self.currency_id = self.env.company.currency_id

    def _with_sale(self):
        return ""

    def _select_sale(self):
        select_ = f"""
            MIN(l.id) AS id,
            l.product_id AS product_id,
            t.uom_id AS product_uom,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.product_uom_qty / u.factor
            * u2.factor) ELSE 0 END AS product_uom_qty,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.qty_delivered / u.factor *
            u2.factor) ELSE 0 END AS qty_delivered,
            CASE WHEN l.product_id IS NOT NULL THEN SUM((l.product_uom_qty -
            l.qty_delivered) / u.factor * u2.factor) ELSE 0 END AS qty_to_deliver,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.qty_invoiced / u.factor *
            u2.factor) ELSE 0 END AS qty_invoiced,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.qty_to_invoice / u.factor *
            u2.factor) ELSE 0 END AS qty_to_invoice,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.price_total
                / {self._case_value_or_one('s.currency_rate')}
                * {self._case_value_or_one('currency_table.rate')}
                ) ELSE 0
            END AS price_total,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.price_subtotal
                / {self._case_value_or_one('s.currency_rate')}
                * {self._case_value_or_one('currency_table.rate')}
                ) ELSE 0
            END AS price_subtotal,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.untaxed_amount_to_invoice
                / {self._case_value_or_one('s.currency_rate')}
                * {self._case_value_or_one('currency_table.rate')}
                ) ELSE 0
            END AS untaxed_amount_to_invoice,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.untaxed_amount_invoiced
                / {self._case_value_or_one('s.currency_rate')}
                * {self._case_value_or_one('currency_table.rate')}
                ) ELSE 0
            END AS untaxed_amount_invoiced,
            COUNT(*) AS nbr,
            s.name AS name,
            s.date_order AS date,
            s.state AS state,
            s.invoice_status as invoice_status,
            s.partner_id AS partner_id,
            s.user_id AS user_id,
            s.company_id AS company_id,
            s.campaign_id AS campaign_id,
            s.medium_id AS medium_id,
            s.source_id AS source_id,
            t.categ_id AS categ_id,
            s.pricelist_id AS pricelist_id,
            s.analytic_account_id AS analytic_account_id,
            s.team_id AS team_id,
            p.product_tmpl_id,
            partner.commercial_partner_id AS commercial_partner_id,
            partner.country_id AS country_id,
            partner.industry_id AS industry_id,
            partner.state_id AS state_id,
            partner.zip AS partner_zip,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(p.weight * l.product_uom_qty
            / u.factor * u2.factor) ELSE 0 END AS weight,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(p.volume * l.product_uom_qty
            / u.factor * u2.factor) ELSE 0 END AS volume,
            l.discount AS discount,
            CASE WHEN l.product_id IS NOT NULL THEN SUM(l.price_unit *
            l.product_uom_qty * l.discount / 100.0
                / {self._case_value_or_one('s.currency_rate')}
                * {self._case_value_or_one('currency_table.rate')}
                ) ELSE 0
            END AS discount_amount,
            concat('sale.order', ',', s.id) AS order_reference"""

        additional_fields_info = self._select_additional_fields()
        template = """,
            %s AS %s"""
        for fname, query_info in additional_fields_info.items():
            select_ += template % (query_info, fname)

        return select_

    def _case_value_or_one(self, value):
        return f"""CASE COALESCE({value}, 0) WHEN 0 THEN 1.0 ELSE {value} END"""

    def _select_additional_fields(self):
        """Hook to return additional fields SQL specification for select part of the
```

```python
            table query.

            :returns: mapping field -> SQL computation of field, will be converted to '_
            AS _field' in the final table definition
            :rtype: dict
            """
            return {}

        def _from_sale(self):
            return """
                sale_order_line l
                LEFT JOIN sale_order s ON s.id=l.order_id
                JOIN res_partner partner ON s.partner_id = partner.id
                LEFT JOIN product_product p ON l.product_id=p.id
                LEFT JOIN product_template t ON p.product_tmpl_id=t.id
                LEFT JOIN uom_uom u ON u.id=l.product_uom
                LEFT JOIN uom_uom u2 ON u2.id=t.uom_id
                JOIN {currency_table} ON currency_table.company_id = s.company_id
                """.format(
                currency_table=self.env['res.currency']._get_query_currency_table(self.env
                .companies.ids, fields.Date.today())
                )

        def _where_sale(self):
            return """
                l.display_type IS NULL"""

        def _group_by_sale(self):
            return """
                l.product_id,
                l.order_id,
                t.uom_id,
                t.categ_id,
                s.name,
                s.date_order,
                s.partner_id,
                s.user_id,
                s.state,
                s.invoice_status,
                s.company_id,
                s.campaign_id,
                s.medium_id,
                s.source_id,
                s.pricelist_id,
                s.analytic_account_id,
                s.team_id,
                p.product_tmpl_id,
                partner.commercial_partner_id,
                partner.country_id,
                partner.industry_id,
                partner.state_id,
                partner.zip,
                l.discount,
                s.id,
                currency_table.rate"""

        def _query(self):
            with_ = self._with_sale()
            return f"""
                {"WITH" + with_ + "(" if with_ else ""}
                SELECT {self._select_sale()}
                FROM {self._from_sale()}
                WHERE {self._where_sale()}
                GROUP BY {self._group_by_sale()}
                {")" if with_ else ""}
            """

        @property
        def _table_query(self):
            return self._query()


```