Reusable Python OO Modules and Webtools for Leverage Buyout and Liquidity Analysis

KEAN3.0 DESIGN & DEVELOPMENT DOCUMENT

Chang Liu

KINDLE ENERGY LLC 500 Alexander Park Dr, Princeton NJ

Introduction

What is KEAN3?

KEAN3 is a fully redesign of both database structure and python codes for KEAN. Using Object-Oriented programming. KEAN is using functional programming which is viewing business logic as mathematical methods. Use data as input and result as output of each standalone function. Develop faster but with limited reusability and extensibility. OOP is viewing real finance world as objects, methods to be 'actions/operations' of the object and data to be 'attributes/properties' of the object. OOP abstracts similarities among objects, and mimics how real-world they interact with each other. OOP requires more time on design phase but increases the reusability of modules. Much easier to maintain and to extend when we need to model new things.

Why we want to do KEAN3? Goals and objectives.

As we add more and more things to KEAN, it increases the complexity of maintenance to functional programming structures. Adding customizations and reducing the reusability. Doing KEAN3, giving us the chance to do an overview of all the tasks we had and challenges we are facing. Let us rethink about the similarities of financial instruments, portfolios, even models. We have following objectives in KEAN3 development,

1. To understand, classify and abstract every financial instrument
2. To map and store our current financial information
3. To build financial models so to solve financial problems
4. To build standard python libraries/modules that can be used by other projects
5. To build configurable ways so to accelerate the processing of financial models
6. To build webtools that can be used by other team members to run models and generate reports

Process Flow

KEAN3 has designed a new way of running models. A configurable, modulized and flexible way. Making the code development less customized so more reusable.

Process of a typical Liquidity/LBO run,

1. Define scenario master information to indicate where to get all the required configurations
2. Define capital structure of a portfolio, all financial components included in a portfolio. Preparing information for financial objects as the following,
    I. Operating Company, which has all the data for Revenue, EBITDA, Capex and working captical
    II. Debt related objects, term loans, revolvers, swaps
    III. Equity object, for equity cash sweep
    IV. TaxRegister, for tax distribution and related activities

3. Define waterfall, the cash flow priorities. Orders of each tranche to get their payment, and define the way how they get them
   A sample for Lightstone liquidity waterfall,

portfolio	scenario	version	level	sub_level	instrument	item	method	direction
Lightstone	2020 CL	v1	1	1	OpCo	EBITDA	normal	inflow
Lightstone	2020 CL	v1	1	2	OpCo	Capex	normal	outflow
Lightstone	2020 CL	v1	1	3	OpCo	working capital	normal	inflow
Lightstone	2020 CL	v1	1	4	OpCo	other cash use	normal	outflow
Lightstone	2020 CL	v1	2	1	Revolver	interest expense	required	outflow
Lightstone	2020 CL	v1	2	1	Swap	interest expense	required	outflow
Lightstone	2020 CL	v1	2	1	TLC	interest expense	required	outflow
Lightstone	2020 CL	v1	2	1	TLB	interest expense	required	outflow
Lightstone	2020 CL	v1	3	1	Revolver	draw	exclusive	inflow
Lightstone	2020 CL	v1	3	1	Revolver	repay	exclusive	outflow
Lightstone	2020 CL	v1	4	1	TaxRegister	ptd	normal	outflow
Lightstone	2020 CL	v1	5	1	TLB	prepayment	exclusive	outflow
Lightstone	2020 CL	v1	5	2	TLB	upsize	exclusive	inflow
Lightstone	2020 CL	v1	6	1	TLC	prepayment	exclusive	outflow
Lightstone	2020 CL	v1	6	2	TLC	upsize	exclusive	inflow
Lightstone	2020 CL	v1	7	1	TLC	amortization	pari passu	outflow
Lightstone	2020 CL	v1	7	1	TLB	amortization	pari passu	outflow
Lightstone	2020 CL	v1	8	1	TLB	dsra release	normal	outflow
Lightstone	2020 CL	v1	8	2	TLC	dsra release	normal	outflow
Lightstone	2020 CL	v1	9	1	Equity	sweep	normal	outflow

OO Design

Modules/Libraries

Scenario_control

To do the overall control of inputs and outputs. Keep the information of versions, key time tags and important dates.

Core classes

Class	Scenario
Attributes	- module: category/model of data this scenario belongs to - table: table this data is being stored in - portfolio - scenario - version
Methods	- print_scenario() - __str__()

Class	ScenarioMaster
Attributes	- outputScenario - startYear: the year that this model starts with - numberOfYears: number of years we want to run on this model - forecastStartMonth: starting point of the forecast section if we have historicals in this model - valuationDate: curve date - inputScenarios: a list of Scenario objects that relate to input data - inputScenarioMasters: a scenario master object can have other complex scenario master objects as inputs (e.g in AMR we have prior forecast and budget which are another AMR forecast that holds a list of inputs)
Methods	- load_sm_fromdb(): load an existing scenario master from database - load_scenario_datetime_fromdb(): load date time information from database for existing scenario master object - build_actuals_period() - build_forecast_period() - save(): to update/insert scenario master object to database

Financial

Combined by two core engines. Dispatch and Financial engines that generate revenue items, fixed costs items, capex items to get to the EBITDA, the starting point of cashflow. This module is not yet fully redesigned in OOP. We still have an independent dispatch model and financials logic to get numbers for EBITDA.

Core Classes

Class	FSLI (Potentially if any fsli that has an independent logic to calculate its values, that fsli will be a subclass inherited from FSLI and maintain its own functions to do calculations, e.g Labor will be inherited from FSLI with implemented methods that calculates labor expense from census)
Attributes	- name - dateStart - dateEnd - amount - creditSign - isSubtotal
Methods	- calc_subtotal(fslis): if a fsli is a subtotal line, pass a list of other fslis that gets to this subtotal item

Liquidity

Cash flow and liquidity analysis module. Abstractions from financial world.

Core classes

Class	OperatingCompany
Attributes	- portfolio - financialsScenario - financialsVersion - financialsTable - ebitda: dictionary for ebitda values - capex: dictionary for capex values - workingCapital: dictionary for working capital numbers - otherCashUse: dictionary for other cash use numbers
Methods	- __get_financials(): private, get financials from database - get_entity_capex() - build_cfo: build cash flow from operations using object's ebitda and capex values

Class	Debt
Attributes	- instrumentID - issueDate, - maturityDate, - term - initialBalance - interestStartDate - amortStartDate - periodicityMonths - annualScheduledAmort - minCashReservePrepay - dayCount - sweepPercent - dsraMonths - oids: a list of OID objects - dfcs: a list of DFC objects - oidPayments - dfcPayments - upsizes - prepays - effectiveInterestRates - interestPayments - requiredDSRAs - dsraCashMovement - amortization - principalBalances - flagPrepayable - flagHistoricals - flagDsraFundByLC
Methods	- __build_period_list (): private, build a list of periods from issue_date to maturity_date - build_principle_balances(): build monthly balances from upsizes and prepays - build_interest_payments(): build monthly interest payments from monthly principle balances - calculate_interest_expense(forecast_month): class method, calculate interest expense for a specific month - build_dsras(): build quarterly required dsras and cash movement if dsra not funded by LC - set_historical_amortization(forecast_start_month, debt_activity_df) - set_historical_size_change(forecast_start_month, debt_activity_df): historical prepays and upsizes - set_historical_interest_payments(forecast_start_month, debt_activity_df)

Class	FixedDabt Inherit from Debt
Attributes	- fixedRate
Methods	- set_effective_interest_rates()

Class	FloatingDebt Inherit from Debt
Attributes	- margins - index: interest rate index
Methods	- set_effective_interest_rates(index_df, forecast_start, floor)

Class	Revolver Inherit from FloatingDebt
Attributes	- creditLine: maximum capacity of revolver
Methods	- build_revolver_draw(ending_cash_balances) - set_historical_revolver_change(forecast_start_month) - set_projected_revolver_change(forecast_start_month, scenario_assumptions_df)

Class	OID
Attributes	- balance - beginDate - endDate - oidDiscount
Methods	- balance_accretion(balance, oid_discount, oid_ytm, begin_date, end_date) - __oid_ytm_calc_wrapper(oid_ytm, *args) - build_monthly_oid_payments() - calc_monthly_oid_payments(balance, begin_date, end_date, oid_discount): static method

Class	Swap
Attributes	- portfolio - instrumentID - index - tradeDate - counterparty - swapRates: list of swap rates information
Methods	- build_swap_interest_payments(index_df) - get_swap_rates_from_db() - build_swap_payments_by_month(start_month, end_month)

Class	TaxRegister
Attributes	- portfolio - effectiveTaxRate - taxSplitRatio - paidTax
Methods	- get_paid_tax_from_db(as_of_date) - calculate_tax_payment(year, total_oid, total_ebitda, total_dfc, total_tax_depreciation, total_interest_expense)

Class	FixedAsset
Attributes	- portfolio - entityName - depreciationMethod - depreciationTerm - inServiceYear - initialPurchasePrice - capex - depreciationAdjustment
Methods	- calcualte_tax_depreciation(additional_capex, year)

LBO

Module for leverage buyout analysis. This module will import from Liquidity and Financial module to accomplish certain tasks.

Class	FreeCashFlow
Attributes	- discountRate - timeZero - discountedAmount - discountFactor
Methods	- calculate_discount_factor - calculate_discounted_cashflow - calculate_wacc(equity_cost_of_capital, debt_cost_of_capital, equity_percentage): static method

Report_writer

Module responsible for writing reports.

KAT/KAD 3.0

Module for webtools