**Documentation About the Model**

1. Naming convention

Although the model is written in python, I think hungarian notation is more suitable for this project. Of course if you have any ideas about naming conventions, it is good to discuss with me. Also please note that there may be some places that such rules are not used, it is because I don’t have the full picture of the model at the beginning. This model will surely be modified a lot later, or even be rewritten. These inconsistencies and bad names will be changed later.

f: function

c: class method

d: dictionary

od: ordered dictionary

t: tuple

l: list

scr:Script (a class defined by me)

var: Variable (a class defined by me)

auto: variant type (can be any type)

sig: inspect.signature

barg:inspect.boundarguments

nt: named tuple

fdec: function decorator

m: module

s: string

i: integer

db:double

df: pandas.dataframe

Also, ds means a dictionary containing strings, dds means a dictionary of dictionaries containing strings. Same for ls, ts, lls and so forth.

Moreover, l\_s means the variable may be list or string

1. Run the model

(I use Spyder). Run mProcess.py, you should get 2 files after the run in the folder containing the scripts: “second.xlsx”, “one.xlsx”

Currently, there is nothing related to actuarial science or finance written. I want to first make sure that the structure of the model is ready for mass calculations.

1. structure of the model

The comments of the model are very messy. You will know all the necessary information about the model in this documentation.

* 1. ActModel

“ActModel” should be the class that interact with users (which may be called an interface) directly. Currently there is no code for this class.

* 1. Process

“Process” is the class that manage all the internal calculations of this model.

code (mProcess.py):

self.\_Input=Input()

self.\_GlobalSettings=GlobalSettings(self.\_Input)

self.\_Output=Output(self.\_Input,self.\_GlobalSettings)

“Process” will create an instance of “Input” Class, which manages all the inputs to the model. It will create “GlobalSettings” and “Output”. “Output” manages all the outputs of the model. I will explain its member functions later.

* 1. Input

Input class reads all the available tables, including (some of these names may not be that good)

* + 1. MPFTable
    2. GenTable
    3. Script
    4. AccumTable
    5. ReportVarTable
    6. OutputFormatTable
    7. KeyGenTables

Input class will first call its “fodLoadAllInputs” to load the tables (this function will load all the tables for one specific type of table), then store them in “dAllInputs” which is actually a dictionary of ordered dictionary of Tables.

console (i is an instance of Input class):

In [3]: i.dAllInputs

Out[3]:

{'MPFTable': OrderedDict([('MPFguy', <mMPFTable.MPFTable at 0xfc17b38>),

('MPFgu3y', <mMPFTable.MPFTable at 0xfc17c50>)]),

'GenTable': OrderedDict([('oneguy', <mGenTable.GenTable at 0xfc17a58>),

('oneguffy', <mGenTable.GenTable at 0xee51fd0>)]),

'Script': OrderedDict([('oneguy333', <mScript.Script at 0xfc1e9e8>),

('oneguy', <mScript.Script at 0xfc1e390>),

('OUTPUT\_CRITERIA', <mScript.Script at 0xfc1ec88>)]),

'KeyGenTable': {'ProdTable': <mKeyGenTable.KeyGenTable at 0xfc17668>,

'ScenTable': <mKeyGenTable.KeyGenTable at 0xfc1e630>,

'GlobalTable': <mKeyGenTable.KeyGenTable at 0xee46b38>},

'AccumTable': OrderedDict([('Accum1',

<mAccumTable.AccumTable at 0xee46fd0>)]),

'ReportVarTable': OrderedDict([('report1',

<mReportVarTable.ReportVarTable at 0xee46c18>)]),

'OutputFormatTable': OrderedDict([('out1',

<mOutputFormatTable.OutputFormatTable at 0xee42a58>)])}

* 1. InputPrototype

Abstract class for all Tables and Scripts. Currently, it has no instances.

Apart from the python scripts, the main folder also contain some sub-folders called script, test\_output, Prod, test\_input, MPF. In these folders, there are some XML files (.txt). Each InputPrototype will read data from 1 single XML file.

InputPrototype::\_fXMLFindChildOutputList function will read any data from the XML file except the “BODY” node.

* 1. Table

inherited from InputPrototype. Currently, it is the same as InputPrototype. I am considering to delete this class later.

* 1. GenTable (generic table) (inherited from Table)

This name is from my software, which may be changed in the future. It will read XML in “test\_input” folder. Currently, it will recognize those columns that can be regarded as “keys” and other columns as “values”. The “BODY” member is a dictionary. I am thinking of changing this self.BODY to pandas.dataframe instead.

* 1. KeyGenTable (inherited from GenTable)

Add some restrictions, s.t. there can only be 1 key column, also the name of this key column will be double checked by assigning values to “sRestriction”

Later, may change self.BODY to dataframe.

* 1. MPFTable (inherited from table)

It does not have key column. Basically, each row is a record for one policy. MPF stands for model point file, which is an actuarial language for “records for customers”. So in real life, you may expect there are more than 10000 rows of records, containing information different policies. There maybe more than 1 MPFTable read by the model, as these MPF Tables may be classified by different products.

Currently, self.BODY is a pandas.dataframe

* 1. Script (inherited from InputPrototype)

A class to read scripts written by actuaries. These scripts (eg .\script\ Script2.txt) containing definitions for different functions that will dependent on each other. Each variable in the script will be stored as a Variable class instance.

* 1. Variable

This class will hold definitions for variables. During initialization, they only contain strings for their function definitions (self.sFunction). Later “self. fSetfFunction” will be called and Variable::\_fFunction will contain a real function definition. Variable class can be called, and have caches to store any calculated values.

I think calling a user-defined class may not be efficient in terms of computation time. We may need to think of a new way to store caches and compute quickly.

* 1. mVarNameSpace

this module is created inside the “Process” class instance. It is used as a namespace for all the Variables (I cannot think of other better ways to do this unfortunately). After mVarNameSpace been set up ,all the “Variable::\_fFunction” will be loaded inside the module. After that, Variable with the same name as their own function will be loaded, so as to replace their functions and be called directly. Also, Process’s Input/output/ GlobalSettings are also loaded in the mVarNameSpace, so Variables can get information from Input, and send results to the Output class instance.

* 1. Process::fRunModel()

It will call fRunModel from mVarNameSpace Class.

Code:

def fRunModel():

odVarRanges=GLOBAL\_SETTINGS.odDimVarRanges

lsVars=GLOBAL\_SETTINGS.odReportVars["ok"]

dNEXT\_POLICY()#test only

for sVar in lsVars:

odTemp=collections.OrderedDict({sKey:odVarRanges[sKey] for sKey in globals()[sVar].lsFuncArgs})

lDimVarsAllProds=list(itertools.product(\*[autoTemp for autoTemp in odTemp.values()]))

dArgs={}

for tTemp in lDimVarsAllProds:

for iInd,sArg in enumerate(odTemp.keys()):

dArgs[sArg]=tTemp[iInd]

globals()[sVar](\*\*dArgs)

OUTPUT.fRecordPolResults(globals()[sVar],odCURR\_OUTPUT\_FORMAT\_RAW\_CHECK())

To help you better understand this code. I have run the following codes in the console.

Console:

* + 1. In [4]: p

Out[4]: <\_\_main\_\_.Process at 0x9d15dd8>

comment: p is a process instance

* + 1. In [5]: p.mVarNameSpace.GLOBAL\_SETTINGS.odDimVarRanges

Out[5]: OrderedDict([('t', range(0, 100)), ('k', range(3, 500)), ('s', range(4, 6))])

comment: this OrderedDict contains range of parameters that will be put in selected “Variable”s. In other words, for a function called f1(t,k,s), we need to calculate

for a function called f2(s,t), we need to calculate

these results1 and results2 should be stored in caches.

* + 1. In [6]: p.mVarNameSpace.GLOBAL\_SETTINGS.odReportVars["ok"]

Out[6]: ['aa', 'bb', 'CC']

comments: the variables that we are interested in . Only these Variable’s values are reported. They are stored in script\Script2.txt. See 2.14 for more details.

* + 1. dNEXT\_POLICY()

The code for this variable in mVarNameSpace is in the following:

@fdecCreateVariable

def dNEXT\_POLICY():

return INPUT.fd\_sNextMPFRow()

INPUT.fd\_sNextMPFRow() is a function that will move the “pointer”(I don’t know how it is called in database terminology) 1 record downwards. By calling this function, the model will jump to the next MPF (policy) record. If the current MPF file reaches an end, it will jump to the next MPF file. For each new record, we need to calculate all the selected variables for the new record again, and store them. The loop continues until there is no records left.

The current way of reading MPF records are highly inefficient and incompatible to parallel running of a large number of model points (policies). I am thinking of creating batches for MPF records and run the model in a parallel way.

code:

odTemp=collections.OrderedDict({sKey:odVarRanges[sKey] for sKey in globals()[sVar].lsFuncArgs})

comment:

globals()[sVar] are defined “Variables” that are also selected to be reported.

globals()[sVar].lsFuncArgs are their arguments (eg, t,k,s)

2.12.6.

code:

lDimVarsAllProds=list(itertools.product(\*[autoTemp for autoTemp in odTemp.values()]))

comment:

Cartesian product of OrderedDict([('t', range(0, 100)), ('k', range(3, 500)), ('s', range(4, 6))])

* 1. AccumTable (inherited from Table)

Each Accumulation is a list of products. There are many products sold by an insurance company. Some of these products share similar characters and can be grouped into one. This group is named as Accumulation.

* 1. ReportVarTable(inherited from Table)

As there are hundreds of Variables in a normal actuarial model, we definitely don’t want to know the values for all of them. Therefore, this table is used to store information about the grouping of Variables.

e.g. in test\_output\ REPORTVAR.txt

<REPORT\_VARS name="ok">

aa,bb,CC

</REPORT\_VARS>

this means that a REPORT\_VARS called “ok” is a group of 3 variables, namely “aa”, “bb” and “CC”. 2.12.3 actually uses this Variable group. You may think of a better name for this class.

* 1. OutputFormatTable (inherited from Table)

it is to read data from “test\_output\ out1.txt”

from the XML code, you can see there are 2 OUTPUT\_FORMATs(you may give better names for it). Each OUTPUT\_FORMAT may have various dimensions.

The basic logic is shown in the following. For example, a person may be classified by sex or the product they buy. The first dimension is “sex” and the second dimension is “product”. After calculating all the Variables for a single policy (for example we have calculated the premium he has paid), we need to add this value to the result. We add his premium (for t =0 to 99) to the category called (male, any\_product\_he\_buy). As sometimes actuaries do not care about individual policies, and only want results aggregated by category. After added to the category he belongs to, the results for calculating his records are discarded.

* 1. VarNameSpace:: fRunModel

the last line of this function is: OUTPUT.fRecordPolResults(globals()[sVar],odCURR\_OUTPUT\_FORMAT\_RAW\_CHECK())

To know the idea better (the function is named badly)

console:

In[7]: p.mVarNameSpace.odCURR\_OUTPUT\_FORMAT\_RAW\_CHECK()

Out[7]:

OrderedDict([('one',

{('first', 'ACCUMULATION', 'ACC1'): True,

('first', 'PRODUCT', 'PROD2'): False,

('second', 'CRITERIA', 'bTemp'): True}),

('second',

{('first', 'ACCUMULATION', 'ACC1'): True,

('first', 'PRODUCT', 'PROD2'): False,

('second', 'CRITERIA', 'bTemp'): True})])

comment: this is basically a check over whether the current policy fulfills the criteria listed in OutputFormatTable.

In[8]: p.mVarNameSpace.dCURR\_POLICY()

Out[8]: {'SONE': 'PRODuu1', 'PRODUCT': 'PROD1', 'STHIRD': 3}

comment: as you can see the current policy’s PRODUCT is PROD1, which is included in the Accumulation called “ACC1”. So, ('first', 'ACCUMULATION', 'ACC1'): True.

Its PRODUCT is not equal to 'PROD2'. So ('first', 'PRODUCT', 'PROD2'): False.

“CRITERIA” is a key word, indicating that “bTemp” is a boolean function which is defined inside mVarNameSpace. Please check “script\ scrOutputCriteria.txt”, we find that “bTemp” will always returns True.

* 1. Output

the output class contains a member called ddReports, which is a dictionary of dictionaries of ResultBlock instances.

In[15]: p.mVarNameSpace.OUTPUT.ddReports["one"][('t',)]

Out[15]: <mResultBlock.ResultBlock at 0x9dd9e48>

comment:["one"] is the key of Output Format, [('t',)] is the key of arguments. Basically, Variables with same arguments are stored in the same ResultBlock.

* 1. ResultBlock

Each tab in a excel output file (eg. “one.xlsx”) represents a ResultBlock. As you can see:



ResultBlock::dfResults is a dataframe, with 2 key columns (the first key column is a tuple of OutputFormat’s dimensions; the second key column is a tuple of parameters). I am looking for better ways to store and represent these data.

The function to change section 2.16:

OrderedDict([('one',

{('first', 'ACCUMULATION', 'ACC1'): True,

('first', 'PRODUCT', 'PROD2'): False,

('second', 'CRITERIA', 'bTemp'): True}),

('second',

{('first', 'ACCUMULATION', 'ACC1'): True,

('first', 'PRODUCT', 'PROD2'): False,

('second', 'CRITERIA', 'bTemp'): True})])

to the following format:

In[19]: p.mVarNameSpace.OUTPUT.\_OutputFormatTable.fodOutputFormatsFromRawToCooked(p.mVarNameSpace.odCURR\_OUTPUT\_FORMAT\_RAW\_CHECK())

Out[19]:

{'one': {

(('first', 'ACCUMULATION', 'ACC1'),('second', 'CRITERIA', 'bTemp')): True,

(('first', 'PRODUCT', 'PROD2'), ('second', 'CRITERIA', 'bTemp')): False},

'second':

{(('first', 'ACCUMULATION', 'ACC1'),('second', 'CRITERIA', 'bTemp')): True,

(('first', 'PRODUCT', 'PROD2'), ('second', 'CRITERIA', 'bTemp')): False}}

is done by a function called fodOutputFormatsFromRawToCooked (I will change this name) from OutputFormat Class. The logic for this function is that, only the results for both dimensions return true, will their combination returns true. The result of Out[19] will then be used to determine which category (eg male, product name) of results will be added from the current policy. (the final part of logic is not yet included in the model, currently, the model just add a column for each variable, also the value is 0 if Out[19]’s values are false).

* 1. Process::fPrintAllReports()

a method to generate “one.xlsx” and “second.xlsx”.