Demonstration of waterfall computer program

This is an annotation of demonstration.py, which illustrates a Python program that reads an XML file for an asset backed securities transaction and produces a table for one of its classes based on an assumed constant prepayment speed, sometimes referred to as a decrement table.

Importing standard libraries

The Python program does not automatically load every feature available and third party features must be explicitly imported as needed. These commands bring in the features used. The first line is a comment. Text following a checkmark # is not interpreted by the program.

```
# Obtain various standard helper functions and classes
from __future__ import division        # needs to be first line
import sys
import os
import plac
import urllib2
from collections import defaultdict
from datetime import date
from datetime import datetime
from dateutil.relativedelta import *
from lxml import etree
from StringIO import StringIO
```

Message to briefly to explain purpose of program

```
help_message = '''
demonstration: calculate a decrement table for Sequoia 2010-H1 at a constant prepayment
rate assumption modified so that each loan that prepays does so in full, rather than a
curtailment.
Usage: python ./demonstration.py cpr where cpr is a decimal fraction between 0.01 and
1.00, inclusive
'''
```

Transaction specific constants

These could also be placed in a separate file, which is better practice. They are included here so that the program would be entirely self-contained.

```
'''Constants, from Sequoia Mortgage Trust 2010-H1 (http://goo.gl/I9Wi)'''
dealname = 'Sequoia 2010-H1'
bond = 'Class A-1'
replinefile = '[dead link]'
margin = 2.25       # identical for each loan
index = 0.9410      # assumed constant per 'modeling assumptions'
expfee = 0.2585     # servicing and trustee fees
reset = margin + index - expfee # interest rate calculation on
                                    # adjustment dates
pbal = 237838333.0  # initial aggregate principal balance of the loans
obal = 222378000.0  # initial aggregate principal balance of the Class A-1
srpct = obal/pbal   # initial Senior Principal Percentage
cod = date(2010,5,1)# cut-off date
close_month = cod - relativedelta(months=1)
anniversary_month = (cod - relativedelta(months=1)).strftime('%B')
'''stepdown dates'''
stepdown = dict(
stepone = [date(2017,5,1), 1.0],
steptwo = [date(2018,5,1), 0.7],
stepthree = [date(2019,5,1), 0.6],
```

```
stepfour = [date(2020,5,1), 0.4],
stepfive = [date(2021,5,1), 0.2]
)
tttdate = date(2013,5,1) # two times test date
num_replines = 16
num_loans = 255
speeds = [0, 0.1, 0.2, 0.3, 0.4, 0.5]

url='[dead link]'
#XML file of loans
```

Helper functions

Functions in Python need to be defined before they are first used. As a general rule, the most generic part of the program occurs first and the most specific part comes toward the end. The first function is known as a generator (because of the yield keyword). Its purpose is to deliver some collection of items one at a time as needed. The second function helps to build dictionaries. A dictionary is a data type that associates keys with values in the same way that dictionary entries have definitions. Given the name of a dictionary, say D, a key, say principal, you can look up a value. The >>> sign indicates an interactive Python prompt, not something typed.

```
>>> D['principal']
246890
>>>
def generateItems(seq):
    for item in seq:
        yield item

def md(lexicon,key, contents):
    """Generic append key, contents to lexicon"""
    lexicon.setdefault(key,[]).append(contents)
```

Helper class

One of the weaknesses of Python for the purposes of the waterfall computer program is that it does not have a large collection of financial functions. To be able to calculate amortization of a loan this class and time value of money function were borrowed. The concepts of class and object in Python can be confusion. An object can be thought of as something that can be acted upon in various ways. A class defines an object and the specific ways it can be used. The Solver class will solve for unknown terms given a function and enough other terms. The tvm function will be provided to Solver, which will then generate the amortizing payment given principal, interest and period.

```
class Solver(object):
    '''takes a function, named arg value (opt.) and returns a
      Solver object
       http://code.activestate.com/recipes/303396/'''
    def __init__(self,f,**args):
        self._f=f
        self._args={}
        # see important note on order of operations in __setattr__ below.
        for arg in f.func_code.co_varnames[0:f.func_code.co_argcount]:
            self._args[arg]=None
        self._setargs(**args)
    def __repr__(self):
        argstring=','.join(['%s=%s' % (arg,str(value)) for (arg,value) in
                            self._args.items()])
        if argstring:
            return 'Solver(%s,%s)' % (self._f.func_code.co_name, argstring)
        else:
            return 'Solver(%s)' % self._f.func_code.co_name
```

```
    def __getattr__(self,name):
        '''used to extract function argument values'''
        self._args[name]
        return self._solve_for(name)
    def __setattr__(self,name,value):
        '''sets function argument values'''
        # Note - once self._args is created, no new attributes can
        # be added to self.__dict__.  This is a good thing as it throws
        # an exception if you try to assign to an arg which is inappropriate
        # for the function in the solver.
        if self.__dict__.has_key('_args'):
            if name in self._args:
                self._args[name]=value
            else:
                raise KeyError, name
        else:
            object.__setattr__(self,name,value)
    def _setargs(self,**args):
        '''sets values of function arguments'''
        for arg in args:
            self._args[arg]  # raise exception if arg not in _args
            setattr(self,arg,args[arg])
    def _solve_for(self,arg):
        '''Newton's method solver'''
        TOL=0.0000001      # tolerance
        ITERLIMIT=1000        # iteration limit
        CLOSE_RUNS=10   # after getting close, do more passes
        args=self._args
        if self._args[arg]:
            x0=self._args[arg]
        else:
            x0=1
        if x0==0:
            x1=1
        else:
            x1=x0*1.1
        def f(x):
            '''function to solve'''
            args[arg]=x
            return self._f(**args)
        fx0=f(x0)
        n=0
        while 1:                     # Newton's method loop here
            fx1 = f(x1)
            if fx1==0 or x1==x0:  # managed to nail it exactly
                break
            if abs(fx1-fx0)ITERLIMIT:
                print "Failed to converge; exceeded iteration limit"
                break
            slope=(fx1-fx0)/(x1-x0)
            if slope==0:
                if close_flag:  # we're close but have zero slope, finish
                    break
                else:
                    print 'Zero slope and not close enough to solution'
                    break
            x2=x0-fx0/slope           # New 'x1'
            fx0 = fx1
            x0=x1
            x1=x2
            n+=1
        self._args[arg]=x1
```

```
        return x1

def tvm(pv,fv,pmt,n,i):
    '''equation for time value of money'''
    i=i/100
    tmp=(1+i)**n
    return pv*tmp+pmt/i*(tmp-1)-fv
## end of http://code.activestate.com/recipes/303396/ }}}
```

The Payoff class

This class does the main work of the program. It uses two lists, a list of selected loan characteristics and a list of the some of the constants. As a class it is used to create a specialized type of object, say O, that will have the logic in the class. A confusing part of the syntax is the term self. It is a placeholder. The class does not know in advance if it will be invoked with an assignment to the name O or the name P or some other name. When it is invoked with the name O, O.bbal would contain the beginning balance for a loan.

```
class Payoff():

    '''prepares a decrement table given constant prepayment speed'''
    def __init__(self, L, C):
        self.L = L
        self.C = C
        self.bbal = float(L[0])     #beginning balance
        self.rbal = self.bbal       #remaining balance
        self.i = float(L[1])        #interest rate in form 4.5
        self.rtm = int(L[2])        #remaining months to maturity
        self.mtr = int(L[3])+1      #months to roll date new i in effect
        self.mta = int(L[4])        #months remaining of interest only
        self.cod = C[0]             #cut-off date
        self.tttdate = C[1]         #twotimes test date
        self.srpct = C[2]           #initial senior percentage
        self.osrpct = C[2]          #original senior percentage
        self.reset = C[3]           #interest rate at reset
        self.stepdown = C[4]        #stepdown dates
        self.pbal = C[5]            #original aggregate principal balance
        self.obal = C[6]            #original aggregate class balance
        self.obsupct = 1 - C[2]     #original subordinate percentage
        s = Solver(tvm,pv=self.bbal, fv=0, i = self.i/12, n = self.rtm)
        self.pmt = s.pmt            #monthly payment
        self.teaser = self.mtr      #counter for initial fixed rate period
        self.io = self.mta          #counter for remaining interest only
        self.n = self.rtm+1         #to take into account range()
        self.current = self.cod + relativedelta(months=+1)
        self.smm = 0.0              #single monthly mortality
    def __nonzero__(self):
        return True
    def __bool__(self):
        return False
    def payone(self):
        def is_twice():             #Twotimes test
            if self.subprct >= 2*self.osubpct:
                return 1
            else:
                return 0
        def is_shrinking():
            if self.srpct > self.osrpct:
                 return 1
            else:
                return 0
        def payoff():
```

```
        import random               #import standard randomization module
        space = int(1//self.smm)    #calculate sample space
        outcomes = [1]              #create list with one positive outcome
        for n in range(space-1):    #for the remainder of the sample space
            outcomes.append(0)      #populate with negative outcome
        payoff = random.choice(outcomes)#randomly choose an outcome
        return payoff                   #report result to calling function
    def senior_prepay_percentage():
        if self.current < self.tttdate and is_twice:
            self.srpppct = self.srpct + 0.5*(1-self.srpct)
        elif self.current >= self.tttdate and is_twice:
            self.srpppct = self.srpct
        elif self.current < self.stepdown['stepone'][0]:
            if is_shrinking():
                self.srpppct = 1.0
            elif is_twice():
                self.srpppct = self.stepdown['stepone'][1]
            else:
                self.srpppct = self.srpppct
        elif self.current < self.stepdown['steptwo'][0]:
            if is_shrinking():
                self.srpppct = 1.0
            elif is_twice():
                self.srpppct = self.stepdown['steptwo'][1]
            else:
                self.srpppct = self.srpppct
        elif self.current < self.stepdown['stepthree'][0]:
            if is_shrinking():
                self.srpppct = 1.0
            elif is_twice():
                self.srpppct = self.stepdown['stepthree'][1]
            else:
                self.srpppct = self.srpppct
        elif self.current < self.stepdown['stepfour'][0]:
            if is_shrinking():
                self.srpppct = 1.0
            elif is_twice():
                self.srpppct = self.stepdown['stepfour'][1]
            else:
                self.srpppct = self.srpppct
        elif self.current < self.stepdown['stepfive'][0]:
            if is_shrinking():
                self.srpppct = 1.0
            elif is_twice():
                self.srpppct = self.stepdown['stepfive'][1]
            else:
                self.srpppct = self.srpppct
        elif self.current >= self.stepdown['stepfive'][0]:
            self.srpppct = self.srpct
        else:
            self.srpppct = self.srpct
        next_month = self.current + relativedelta(months=+1)
        self.current = next_month
    senior_prepay_percentage()  #calculate senior prepayment
                                #percentage
    self.teaser -= 1            #reduce remaining teaser period
    self.io -= 1                #reduce remaining interest only period
    self.bbal = self.rbal       #beginning balance to last period's ending
    ipay = self.rbal*self.i/1200    #interest payment portion
    if payoff():
        self.smm = 1.0
    if self.mta > 0:            #if during interest only period
```

```
            self.paydown = 0        #no scheduled principal
            self.prepay = self.smm*(self.bbal-self.paydown)
        else:
            self.paydown = -self.pmt-ipay # reverse negative paid out conv
            self.prepay = self.smm*(self.bbal-self.paydown)
        if self.rtm > 0:            #decrement remaining term to maturity
            self.rtm -= 1
        if self.mtr == 0:           #begin 12-month reset period 11 .. 0
            self.mtr = 11
        elif self.mtr > 0:          #decrement months to reset
            self.mtr -= 1
        if self.mta > 0:            #decrement months to end of i/o period
            self.mta -= 1
        if self.bbal == 0:          #see if final payment has been made
            self.paydown = 0
            self.prepay = 0
        elif self.bbal >= self.paydown + self.prepay:  #not last payment?
            self.rbal -= self.paydown + self.prepay
        elif self.bbal < self.paydown: # scheduled payment enough to final out
            self.paydown = self.bbal
            self.prepay = 0
            self.rbal = 0
        elif self.bbal < self.prepay: # prepayment enough to final out
            self.paydown = self.bbal
            if self.bbal > 0:       # if any still left, allocate to prepay
                self.prepay = self.bbal
                self.rbal = 0
        else:
            self.rbal = 0
        if self.teaser == 1:        #last month of fixed rate period
            self.i = self.reset     #change interest rate for following month
            s = Solver(tvm,pv=self.rbal, fv=0, i = self.i/12, \
            n = self.rtm+1)         #calculate new amortizing payment
            self.pmt = s.pmt        #set new payment
        if self.io == 1:            #last month of i/o period
            s = Solver(tvm,pv=self.rbal, fv=0, i = self.i/12, \
            n = self.rtm)           #calculate amortizing payment
            self.pmt = s.pmt        #set new payment
        yield self.srpct*self.paydown + self.srpppct*self.prepay
```

Data structures to hold loan records and an index to the records.
These should also be ideally isolated to a separate transaction specific file and imported. The number required depends on the number of loans. It would be nice if it were possible to simply write a function to call these forth as needed, but it's not. The first set of statements creates the series of data dictionaries. The list called websters is composed of each of the dictionaries, so that they can be called in turn. Together with dictionaries, lists are one of the most often used data types in Python. Lists can be thought of as written in pencil with items erasable. A similar construct is a tuple, which is a series of items in ink. That is lists are mutable and tuples are not. Strings, which are a collection of one or more characters, are also an immutable data type.
#create an empty dictionary for each loan record

```
websters = []
for i in range(num_loans):
    websters.append(defaultdict(list))
```

Data collection and preparation

Obtaining the XML data from the internet

This standard function collects the XML from the location specified by url, one of the transaction specific constants above and assigns it the name content.

```
content = urllib2.urlopen(url).read()
```

Picking apart the XML data

The third party lxml module makes organizing the XML data simple. It finds the parent or root of the document tree from which a list of the children nodes, in this case loan records, is produced. Next, the variable lexicon is assigned to the generateItems helper function using the websters list as the source. The remaining lines are two nested loops. The outer loop steps through the records from the XML data, calls up the next in order dictionary and then the inner look steps through the fields in each record, extracts the field name, to use as a key, and text, to use as the value and adds it to that dictionary. This continues until all of the fields in all of the records have been processed.

```
root = etree.fromstring(content)
records = list(root)
lexicon = generateItems(websters)
for record in records:
    lex = lexicon.next()
    for field in record:
       md(lex, field.attrib['name'], field.text)
```

Extracting selected loan data

Not all of the many loan data characteristics are needed for the limited purpose of this program. Many of them relate to credit underwriting and would be of use in assessing likelihood of default. Because the data for each loan is in its own dictionary, the needed subset can be readily assembled. The first line creates an empty list, tape, to hold the results. The next line begins a loop, stepping through each loan in the list of dictionaries. It starts by creating an empty list, record, to hold the results, and then collects the values needed, using statements in the form loan['key']. Because the data in the dictionary was not processed before, some of the fields need to be converted from strings to other data types, floats, integers and dates. An integer is a whole number. A float is a number with a decimal point. Even through 1 and 1.0 are equal, the int type and float type they represent have different properties. The end result is the list of list (tape) of the list of characteristics for each loan.

```
tape = []
for loan in websters:
    record = []
    record.append(float(loan['obal'][0]))
    record.append(float(loan['cintpct'][0]))
    tmat = loan['maturity'][0]
    mat = datetime.strptime(tmat, '%Y-%m-%d').date()
    to_mat = relativedelta(mat,cod)
    mtm = to_mat.months + to_mat.years*12
    record.append(mtm)
    fpd = datetime.strptime(loan['fpd'][0], '%Y-%m-%d').date()
    to_roll = relativedelta(fpd + relativedelta(months=60), cod)
    mtr = to_roll.months + to_roll.years*12
    record.append(mtr)
    intonlyterm = int(loan['intonlyterm'][0])
    to_amort = relativedelta(fpd + relativedelta(months=intonlyterm), cod)
    mta = to_amort.months + to_amort.years*12
    record.append(mta)
    tape.append(record)
```

Using the above to produce the table

The run_loan_payoff function takes a constant prepayment rate and generates a decrement table. The list C collects needed constants, the current balance is set to the original balance and the first anniversary is calculated from the cut-off date.
The line
E = {}
creates an empty dictionary for each loan (record) to hold its Payoff object and the objects are created by the following loop.

The next loop should also not have been hardcoded (the beginning and ending anniversaries 2011 and 2041 should have been taken from constants), which it has been for illustration purposes. For each year in that range, each month in the year, and each loan each month, the loan's Payoff object generates a payment, which can include a prepayment in full based on the roll of the dice represented by the CPR prepayment speed. The result is another list of lists (column) for the years with the payments on each loan. These are then aggregated to get the ending balance for each year, from which the percent outstanding of the target class can be calculated. Finally, the table is printed out.

```
def run_loan_payoff(cpr):
    '''cpr = 0.1 Constant Prepayment Rate in decimal fraction'''
    C = [cod, tttdate, srpct, reset, stepdown, pbal, obal]
    cbal = obal
    anniversary = cod.year+1
    E = {}
    for record in tape:
        md(E,'tape', Payoff(record,C))
    twelfth = 1.0/12.0
    smm = 1.0 - (1.0-cpr)**twelfth  #  single monthly mortality
    column = []                     # empty list to collect principal payments
    for year in range(2011,2041):
        annual = []                 # temporary list
        for month in range(12):
            for entry in E['tape']:
                payment = []        # temporary list
                entry.srpct = srpct # set object senior percentage
                entry.subpct = 1 - srpct
                entry.smm = smm     # set smm for object
                try:                # while still data
                    payment.append(entry.payone().next())
                except StopIteration:
                    pass
                annual.append(sum(payment))    # aggregate for month
                cbal -= sum(payment)           # knock down senior
            sprct = cbal/obal                  # recalculate senior percentage
        column.append(annual)                  # collect months
    column[:] = [sum(item) for item in column] # aggregate for year
    cbal=obal
    ''' output decrement table for given CPR speed '''
    print "%s %s at CPR of %d%%" % (dealname, bond, cpr*100)
    for year in column:
        cbal -= year
        percentout = round(cbal/obal*100,2)
        if percentout >= 1:
            print("%s %d:\t\t%0.0f") % (anniversary_month,  anniversary,\
            percentout)
        elif percentout <= 0:
            print("%s %d:\t\t0") % (anniversary_month, anniversary)
        else:
            percentout < 1
            print("%s %d:\t\t*") % (anniversary_month, anniversary)
        anniversary += 1
```

Running the completed program

There are two ways of running the program.

Shell

The first is from a shell prompt:

```
% python ./demonstration.py 0.10
```

The remaining code deals with that situation. It basically notifies the user if a number does not end the command.

```
def main(cpr_arg):
    print help_message
    cpr = float(cpr_arg) # command line argument is a string
    run_loan_payoff(cpr) # call the function to produce the table

if __name__ == "__main__":
        plac.call(main)
```

The second is within the interpreter

First, the Python interpreter is started

```
% python
>>>
```

Then, the pieces of the program are imported.

```
>>> from demonstration import *
>>>
```

After that, any number of tables at different speeds (0.1, 0.125, 0.2, etc.) can be prepared

```
>>> run_loan_payoff(0.1)
```

This will produce a table *like* the following:

```
Sequoia 2010-H1 Class A-1 at CPR of 20%
April 2011:       78
April 2012:       60
April 2013:       47
April 2014:       35
April 2015:       26
April 2016:       20
April 2017:       16
April 2018:       11
April 2019:       8
April 2020:       5
April 2021:       3
April 2022:       *
April 2023:       0
April 2024:       0
April 2025:       0
April 2026:       0
April 2027:       0
April 2028:       0
April 2029:       0
April 2030:       0
April 2031:       0
April 2032:       0
```

```
April 2033:       0
April 2034:       0
April 2035:       0
April 2036:       0
April 2037:       0
April 2038:       0
April 2039:       0
April 2040:       0
>>>
```

The results are likely to be different each time. This is because whether any given loan pays off in any given month is based on a probability for a given prepayment speed. Thus, at a CPR of 0.1, a loan has an approximately 1 in 113 chance of paying of in any particular month. The determination, each month, of whether a payoff occurs is based on a random selection from the possibilities of payoff (1 chance) or no payoff (112 chances). The function assumes no defaults, similarly to the traditional decrement table. However, a default function based on some combination of loan characteristics could be used to arrive at a probability of default in any given month similarly.

Percentage of Initial Class Principal Amount of the Class A-1 Certificates Outstanding at the Following Percentages of CPR

	Class A-1 Certificates					
Prepayment Assumption	**0%**	**10%**	**20%**	**30%**	**40%**	**50%**
Distribution Date						
Initial Percentage	100	100	100	100	100	100
April 2011	100	89	78	68	57	46
April 2012	99	79	61	45	32	22
April 2013	99	70	47	30	18	10
April 2014	98	62	37	21	11	5
April 2015	98	55	30	15	6	2
April 2016	97	48	24	10	4	1
April 2017	96	43	19	7	2	*
April 2018	96	38	15	5	1	0
April 2019	95	34	12	3	*	0
April 2020	92	30	9	2	0	0
April 2021	89	26	7	1	0	0
April 2022	85	22	5	*	0	0
April 2023	82	19	4	*	0	0
April 2024	78	17	3	0	0	0
April 2025	74	14	2	0	0	0
April 2026	70	12	1	0	0	0
April 2027	65	10	1	0	0	0
April 2028	61	9	*	0	0	0
April 2029	56	7	*	0	0	0
April 2030	52	6	0	0	0	0
April 2031	47	5	0	0	0	0
April 2032	42	4	0	0	0	0
April 2033	37	3	0	0	0	0
April 2034	31	2	0	0	0	0
April 2035	26	1	0	0	0	0
April 2036	20	1	0	0	0	0
April 2037	14	*	0	0	0	0
April 2038	8	0	0	0	0	0
April 2039	2	0	0	0	0	0
April 2040	0	0	0	0	0	0
Weighted Average Life in Years (to Maturity)†	19.61	7.58	4.07	2.59	1.81	1.33
Weighted Average Life in Years (to Call) †**	19.53	7.21	3.78	2.39	1.67	1.23
Weighted Average Life in Years (to Roll) †***	4.29	3.40	2.68	2.11	1.65	1.29

* Indicates a value between 0.0% and 0.5%.

† The weighted average life of a certificate of any class is determined by (i) multiplying the assumed reduction, if any, in the Class Principal Amount on each distribution date by the number of years from the date of issuance of the certificate to the related distribution date, (ii) adding the results, and (iii) dividing the sum by the total principal paid.

** Assuming the mortgage loans are purchased on the first date permitted under "Description of the Certificates—Optional Purchase of the Mortgage Loans."

*** Assuming the mortgage loans prepay in full at their related first interest adjustment date.