Practical Lesson 4

Matteo Sani
matteosan1@gmail.com

October 16, 2019

1 Forward Rates and Classes

1.1 Recap

In the first three lessons we looked at:

- basic Python statements, expressions and objects (lists, dictionaries)
- dates, tuples and functions
- we started looking at how to implement functionality related to the theoretical lessons

1.2 Today's lesson

We're going to look at classes and while we talk about this topic we're going to move ahead with implementing functionality related to the theory lessons.

1.3 Calculating Forward Rates

Last week we wrote a function called `df` for calculating a discount factor at any date, given a set of discount factors each relative to a corresponding pillar date, using log-linear interpolation. Now we want a function to compute forward rates.

The formula to calculate the forward rates can be found exploiting the property that investing at rate r_1 for the period $(0, T_1)$ and then *reinvesting* at rate $r_{1,2}$ for the time period (T_1, T_2) is equivalent to invest at rate r_2 for the time period $(0, T_2)$ (i.e. no arbitrage condition, two investors shouldn't be able to earn money from arbitraging between different interest periods). That said:

$$(1 + r_1 T_1)(1 + r_{1,2}(T_2 - T_1)) = 1 + r_2 T_2$$

Solving for $r_{1,2}$ leads to

$$F(T_1, T_2) = r_{1,2} = \frac{1}{T_2 - T_1}\left(\frac{D(T_1)}{D(T_2)} - 1\right) \quad (\text{where } D(T_i) = \frac{1}{1 + r_i T_i})$$

```
In [ ]: from datetime import date
        import numpy, math

        today_date = date (2019, 1, 1)

        pillar_dates = [date(2019 , 1 ,1),
```

```
                date(2020, 1, 1),
                date(2021, 10 ,1)]
discount_factors = [1.0, 0.97, 0.72]

def df(d):
    log_discount_factors = [math.log(discount_factor) \
                            for discount_factor in discount_factors]
    pillar_days = [(pillar_date - today_date).days \
                   for pillar_date in pillar_dates]
    d_days = (d - today_date).days
    interpolated_log_discount_factor = \
        numpy.interp(d_days, pillar_days, log_discount_factors)

    return math.exp(interpolated_log_discount_factor)

def forward_rate(t1, t2):
    return 365.0/(t2-t1).days * (df(t1) / df(t2) - 1)

forward_rate(date(2019, 2, 1), date(2019, 8, 1))
```

1.3.1 2008 Financial Crisis

Looking at the historical series of the Euribor (6M) rate versus the Eonia Overnight Indexed Swap (OIS-6M) rate over the time interval 2006-2011 it becomes apparent how before August 2007 the two rates display strictly overlapping trends differing of no more than 6 bps.

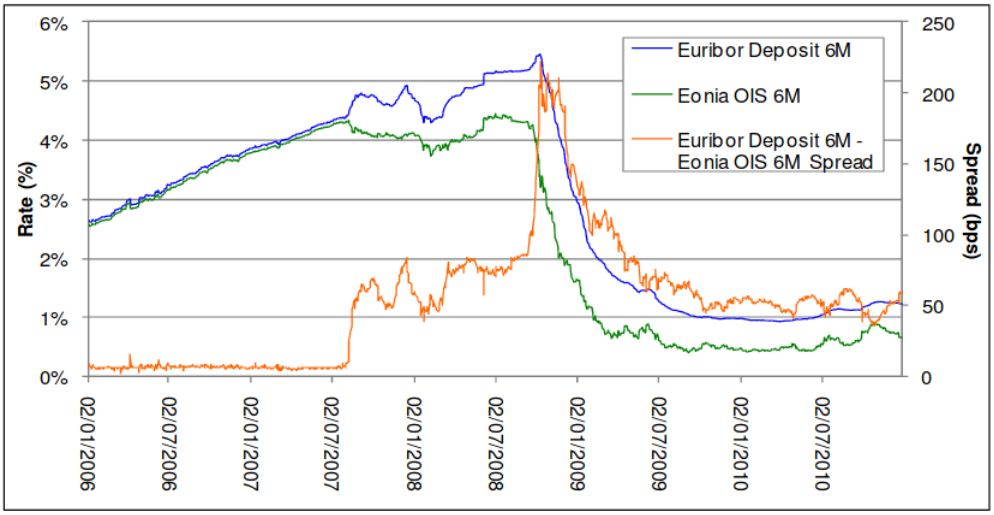


Figure 1: historical series of Euribor Deposit 6M rate versus Eonia OIS 6M rate. The corresponding spread is shown on the right axis (Jan. 06 – Dec. 10 window, source: Bloomberg).

In August 2007 however we observe a sudden increase of the Euribor rate and a simultaneous decrease of the OIS rate that leads to the explosion of the corresponding basis spread, touching the peak of 222 bps in October 2008, when Lehman Brothers filed for bankruptcy protection. Successively the basis has sensibly reduced and stabilized between 40 bps and 60 bps (notice that the pre-crisis level has never been recovered). The same effect is observed for other similar couples, e.g. Euribor 3M vs OIS 3M.

The reason of the abrupt divergence between the Euribor and OIS rates can be explained by

considering both the monetary policy decisions adopted by international authorities in response to the financial turmoil, and the impact of the credit crunch on the credit and liquidity risk perception of the market, coupled with the different financial meaning and dynamics of these rates.

- The Euribor rate is the reference rate for over-the-counter (OTC) transactions in the Euro area. It is defined as "the rate at which Euro interbank Deposits are being offered within the EMU zone by one prime bank to another at 11:00 a.m. Brussels time". The rate fixings for a strip of 15 maturities, ranging from one day to one year, are constructed as the average of the rates submitted (excluding the highest and lowest 15% tails) by a panel of banks 42 banks, selected among the EU banks with the highest volume of business in the Euro zone money markets, plus some large international bank from non-EU countries with important euro zone operations. **Thus, Euribor rates reflect the average cost of funding of banks in the interbank market at each given maturity. During the crisis the solvency and solidity of the whole financial sector was brought into question and the credit and liquidity risk and premia associated to interbank counterparties sharply increased.** The Euribor rates immediately reflected these dynamics and raise to their highest values over more than 10 years. As seen in the plot above, the Euribor 6M rate suddenly increased on August 2007 and reached 5.49% on 10th October 2008.
- The Eonia rate is the reference rate for overnight OTC transactions in the Euro area. It is constructed as the average rate of the overnight transactions (one day maturity deposits) executed during a given business day by a panel of banks on the interbank money market, weighted with the corresponding transaction volumes. **The Eonia Contribution Panel coincides with the Euribor Contribution Panel, thus Eonia rate includes information on the short term (overnight) liquidity expectations of banks in the Euro money market. It is also used by the European Central Bank (ECB) as a method of effecting and observing the transmission of its monetary policy actions. During the crisis the central banks were mainly concerned about restabilising the level of liquidity in the market, thus they reduced the level of the official rates.** Furthermore, the daily tenor of the Eonia rate makes negligible the credit and liquidity risks reflected on it: for this reason the OIS rates are considered the best proxies available in the market for the risk-free rate.

As a practial result, after the 2008 financial crisis, it is not possible anymore to use a single discount curve to correctly price forward rates of all tenors. For example, if we want to calculate the net present value of a forward 6-month libor coupon, we need to simultaneously use two different discount curves:

- the 6-month libor curve for determining the forward rate
- the EONIA curve for discounting the expected cash flow

Essentially we are now going to explore how to implement the following calculation:

$$\text{NPV} = D_{\text{EONIA}}(T_1) \times \frac{1}{T_2 - T_1}\Big(\frac{D_{\text{LIBOR}}(T_1)}{D_{\text{LIBOR}}(T_2)} - 1\Big)$$

Programming problem We only have one `df` function, and one set of data (i.e. the curve inputs): how can we generalize what we have done so far in such a way that we can conveniently do the above calculation ?

One *BAD* idea would be to write a different discount factor function for each curve:

```
# THIS IS PSEUDOCODE JUST FOR ILLUSTRATION
eonia_pillar_dates = [date(2019, 10, 1),
                      date(2020, 10, 1),
                      date(2021, 10, 1)]
eonia_discount_factors = [1.0, 0.95, 0.8]
libor6m_pillar_dates = [date(2019, 10, 1),
                        date(2020, 4, 1),
                        date(2020, 10, 1)]
libor6m_discount_factors = [1.0, 0.98, 0.82]

def df_eonia(t):
    log_discount_factors = [math.log(discount_factor) \
       for discount_factor in eonia_discount_factors]
    interpolated_log_discount_factor = \
       interp_over_dates(t, eonia_pillar_dates,
                         log_discount_factors)
    return math.exp(interpolated_log_discount_factor)

def df_libor(t):
    log_discount_factors = [math.log(discount_factor) \
       for discount_factor in libor6m_discount_factors]
    interpolated_log_discount_factor = \
        numpy.interp(t, libor6m_pillar_dates, log_discount_factors)
    return math.exp(interpolated_log_discount_factor)

def forward_rate(t1, t2):
    return 365.0/(t2 - t1).days * (df_libor(t1) / df_libor(t2) - 1.0)

rate = forward_rate(date(2019, 10, 1), date(2020, 4, 1))
npv = df_eonia(date(2019, 10, 1)) * rate
```

this becomes pretty laborious, since we need to rewrite a function for every discount curve we add (e.g. if we have another currency or more libor rates with different tenors). Also, if we want to change the behaviour of the df function (e.g. change the type of interpolation), we then have to go and change each single implementation (`df_eonia`, `df_libor`, etc...) so it's not a convenient way of organizing the code.

An alternative could be to generalize the `df` function so that it takes the pillar dates and discount factors as arguments, then we pass in the data we want to use to calculate the interpolated discount factor in each specific case:

```
# THIS IS PSEUDOCODE JUST FOR ILLUSTRATION
def df(pillar_dates, discount_factors, t):
    log_discount_factors = [math.log(discount_factor) \
        for discount_factor in discount_factors]
    interpolated_log_discount_factor = \
        interp_over_dates(t, pillar_dates, log_discount_factors)
    return math.exp(interpolated_log_discount_factor)
```

```
def forward_rate(t1, t2):
    return 365.0 / (t2 - t1).days * (
        df(libor6m_pillar_dates, libor6m_discount_factors, t1) /
        df(libor6m_pillar_dates, libor6m_discount_factors, t2) - 1.0)

rate = forward_rate(date(2019, 10, 1), date(2020, 4, 1)) \
    * df(eonia_pillar_dates, eonia_discount_factors, date(2019, 10, 1))
```

This is still very inconvenient though, because we always have to pass a set of static parameters to every function call. It's just about OK as long as the parameter set is small (like in this case with only few items) but as it gets larger it becomes unmanageable.

However, as we have seen, Python allows you to represent collections of objects with dictionaries. A clear improvement could be, instead of passing a long list of data parameters to each function call, to group the datasets into dictionaries, and then pass those to the function.

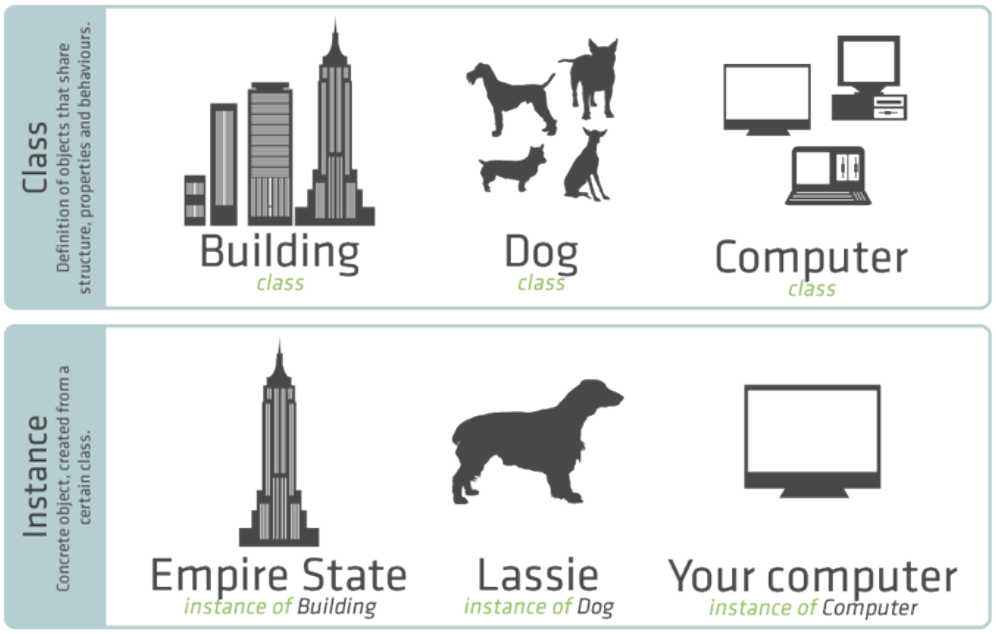
```
# THIS IS PSEUDOCODE JUST FOR ILLUSTRATION
# One generalized df function, using dictionaries for representing datasets

eonia_data = {"pillar_dates": [date(2019, 10, 1),
                               date(2020, 10, 1),
                               date(2021, 10, 1)],
              "discount_factors": [1.0, 0.95, 0.8]
             }
libor6m_data = {"pillar_dates": [date(2019, 10, 1),
                                 date(2020, 4, 1),
                                 date(2020, 10, 1)],
                "discount_factors": [1.0, 0.98, 0.82]
             }
def df(data, t):
    pillar_dates = data['pillar_dates']
    discount_factors = data['discount_factors']
    log_discount_factors = [math.log(discount_factor) \
        for discount_factor in discount_factors]
    interpolated_log_discount_factor = \
        interp_over_dates(t, pillar_dates, log_discount_factors)
    return math.exp(interpolated_log_discount_factor)

def forward_rate(t1, t2):
    return 365.0/(t2 - t1).days * (
        df(libor6m_data, t1) /
        df(libor6m_data, t2) - 1.0)

npv = df(eonia_data, date(2019, 10, 1)) \
    * forward_rate(date(2019, 10, 1), date(2019, 4, 1))
```

This design pattern, i.e. using dictionaries to group together data, and then having functions operate on those dictionaries, perhaps with a few additional parameters, is so useful that Python (and many other programming languages) have a built-in feature that allows you to do this conveniently: **classes**.



Graphical representation of a class instance.

1.4 Classes

Classes are a key ingredient of *Object Oriented Programming* (OOP) and their concept is implemented in many languages like `Python`, `Java`, `C++`. OOP is a programming model in which programs are organized around data, or objects, rather than functions and logic. **An object can be defined as a dataset with unique attributes and behaviour** (examples can range from physical entities, such as a human being that is described by properties like name and birthday, down to abstract concepts as a discount curve). This opposes the historical approach to programming where emphasis was placed on how the logic was written rather than how to define the data within the logic. In this framework classes are a mean for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

Let's summarize here some terminology:

- a class is a collection of related functions, and these are called the *methods* of the class;
- methods act on *instances* of the class;
- an *instance* is basically a collection of related data;
- each data item has a name, and those names are called the *attributes* of the class.

Essentially classes are collections of functions that operate on a dataset, and instances of that class represent individual datasets (or in othre words a specialization of that class).

Class methods always take the instance `self` as the first argument, and fall into two categories:

- normal methods which use or modify the instance attributes;
- special methods, which define the class behaviour: you can spot these because they start and end with two underscores (__).

The `self` argument is very important since it allows a method to use its class attributes.

There are lots of other things you can do with classes, but this is enough for now. Let's take a look at an example:

```
In [ ]: from datetime import date

        # this is the class definition
        # usually classes use camel naming convention
        class Person:

            # the special method __init__ allows to instanciate a class
            # with an initial dataset (in this example a name and a birthday)
            def __init__(self, name, date_of_birth):
                # attribute of the class Person
                # name and self.name are different variable !!!
                # name will be destroyed once __init__ is processed
                # self.name lives with every particualar instance of Person
                self.name = name
                # attribute of the class Person
                self.date_of_birth = date_of_birth

            # this normal method computes the current age of the
            # "instanciated" person
            def age(self):
                today = date.today()
                age = today.year - self.date_of_birth.year
                if today.month < self.date_of_birth.month or \
                    today.day < self.date_of_birth.day:
                    age -= 1
                return age
```

```
In [ ]: # here we instanciate (create an instance of) the class
        # in other words we "specialize" a generic Person with some data

        me = Person("Matteo", date(1974, 10, 20))
        print (type(me))
```

__init__ is the simplest example of special methods, it is called every time a class is instanciated (e.g. when you write me = Person(...)) and initializes the attributes of the class.

```
In [ ]: # to access class attributes you have to use .
        me.name
```

```
In [ ]: me.date_of_birth
```

```
In [ ]: # to call a class method you have to use .
        # passing the parameters if needed
        me.age()
```

```
In [ ]: from datetime import date
```

```
        # let's add a new method to print in a nicer form
        # the age of the person
        class Person:

            def __init__(self, name, date_of_birth):
                self.name = name
                self.date_of_birth = date_of_birth

            def age(self):
                today = date.today()
                age = today.year - self.date_of_birth.year
                if today.month < self.date_of_birth.month or \
                    today.day < self.date_of_birth.day:
                    age -= 1
                return age

            # methods in a class are just functions which can work
            # with the class attributes
            # Remember I told you functions can have no return ?
            def print_age(self):
                print ("{} is {} years old right now"\
                       .format(self.name, self.age()))
```

```
In [ ]: her = Person("Francesca", date(1986, 1, 27))
```

```
In [ ]: her.print_age()
```

1.5 Exercises

1.5.1 Exercise 4.1

Write two classes, `Circle` and `Rectangle` that given the radius and height, width respectively allow to compute area and perimeter of the two shapes. Test them with the following:

```
a_circle = Circle(5)
print ("My circle has an area of {} m**2".format(a_circle.area()))

a_rectangle = Rectangle(3, 6)
print ("My rectangle has a perimeter of {} m and an area of {} m**2" \
    .format(a_rectangle.perimeter(), a_rectangle.area()))
```

1.5.2 Exercise 4.2

Define a class `Songs`, its `__init__` should take as input a dictionary (`lyrics` that contains lyrics line by line). Define a method, `sing_me_a_song` that prints each element of the lyrics in his own line. Also test it with the follwing input.

```
lyrics = {"Wonderwall":["Today is gonna be the day",
                        "That they're gonna throw it back to you",
```

```
                    "By now you should've somehow", "..."],
        "Wish you were here":  ["So, so you think you can tell",
                                "Heaven from hell",
                                 "Blue skies from pain", "..."]}
```

1.5.3 Exercise 4.3

Define a Point2D class that represent a point in a plane. Its `__init__` method should accept the point coordinates `x` and `y`. Write a method `distanceTo` that compute the distance of the point to another passed as input. Test the class by printing the distance of the point $P = (4, 5)$ to the origin $P = (0, 0)$ and to $P = (3, 4)$.

1.5.4 Exercise 4.4

Try to write a `DiscountCurve` class which contains the pillar dates and pillar discount factors as attributes and which has methods for calculating the discount factor and forward rate at arbitrary dates.

Hint:

```
# here goes import statement of the needed modules
import ABCD
from XYZ import xyz

# usually classes have CamelCase naming convention
class DiscountCurve:

    # the special __init__ method defines
    # how to construct instances of the class
    # so you need to identify the attributes you need to store
    # in the class defining a discount curve
    def __init__(self, ...):

    # then we want to add a method to compute the discount
    # factor at an arbitrary value date
    # using the data stored in the instance
    def df(self, param1, param2, ...):
      # the implementation can follow what we did in the
      # function we wrote last week but this time has to
      # use the class attributes

    # finally we want a method to calculates the forward rate
    # based on the discount curve data stored in the instance
    def forward_rate(self, param1, param2, ...):
        # here of course we can use the df method
        # implemented above to calculate the forward rate
```