Week 5 — *Manipulating classes with Python*

The goal of the present exercise is to program a family of objects to compute series and to dump them. We will use Python for this exercise. In Python the concept of interfaces does not exist. You can therefore write both the definition and the implementation of your classes in the same **.py** file. However, by convention this file should be named like the class it contains, *e.g.* dumper_series.py for the class **DumperSeries**.

Exercise 1: *Series class*

1. Create a class named **Series** which will be the mother class for all other series classes. The keyword *pass* ensures that the function can be called without problem even though no implementation is provided. As example the constructor of **Series** does nothing in this case.

   ```
   class Series:
       def __init__(self):
           pass
   ```

   Since this class is not computing any type of series but serves just as an interface for its subclasses the function **compute** cannot be implemented for this series. Instead you can raise an exception when this function is called on an object of type **Series** in order to ensure that no object of this class can be used. The class therefore acts as an abstract class.

   ```
   def compute(self, N):
       raise Exception("pure virtual function")
   ```

   This simple class will represent the interface of the family of classes inheriting from **Series**.

2. Create a class named **ComputeArithmetic** which inherits from **Series** and which implements the converging series:

   $$S_n = \sum_{k=1}^{N} k$$

   in the function

   ```
   def compute(self, N):
       ...
       return series_value
   ```

3. Create (instantiate) a **ComputeArithmetic** object in the **main** and call the **compute** method, to output the result to screen (using print), and test the program.

4. Create another **ComputePi** class which computes the series converging towards *pi*:

   $$\pi = \sqrt{6 \sum_{k=1}^{N} \frac{1}{k^2}}$$

5. Modify the main to decide which of **ComputeArithmetic** or **ComputePi** is to be instantiated.

Exercise 2: ***Dumper class***

1. Create a class named **DumperSeries** with a constructor taking a Series object as a paramter:

   ```
   class DumperSeries:

      def __init__(self, series):
           self.series = series
   ```

   Define for this class a function **dump** which will raise an exception when it is called:

   ```
      def dump(self):
          raise Exception("pure virtual function")
   ```

   Therefore, this simple class will represent the interface of the classes inheriting from **DumperSeries** but no object of the class itself can be created.

2. Create a class named **PrintSeries** which inherits from **DumperSeries**. This class will output (to screen) every step out of frequency that is lower than *maxiter*. Thus frequency and *maxiter* should be declared as members and set by the constructor. Implement the method **dump**.

3. Create (instanciate) a **PrintSeries** object in the **main** and call the **dump** method onto the previously created **Series** reference.

4. In the **Series** class, append the virtual method

   ```
   def getAnalyticPrediction(self):
       raise Exception("pure virtual function")
   ```

   which provides the analytic prediction of a class. Implement the overriding method in the **ComputePi** class.

5. In the **dump** function of the **PrintSeries** class, use the analytic prediction to get the convergence towards the limit.

6. Create another class named **PlotSeries** which also inherits from **DumperSeries**. Implement the method **dump** which will plot numerical results of all steps lower than *maxiter*. It should also plot the analytical prediction.

7. Modify the **main** to decide which of **PrintSeries** or **PlotSeries** is to be instantiated.

Exercise 3: ***Series complexity***

1. Evaluate the global complexity of your program

2. Modify the **Series** class to have the members

   ```
   def __init__(self):
       self.current_term = 0
       self.current_value = 0;
   ```

3. Use these members in **ComputeArithmetic** and **ComputePi** in order to prevent re-computation of the entire series each time a **DumperSeries** would call it. Factorize the code as much as possible (by writing generic behavior in the **Series** class directly)

4. How is the complexity now ?

5. In the case you want to reduce the rounding errors over floating point operation by summing terms reversely, what is the best complexity you can achieve ?