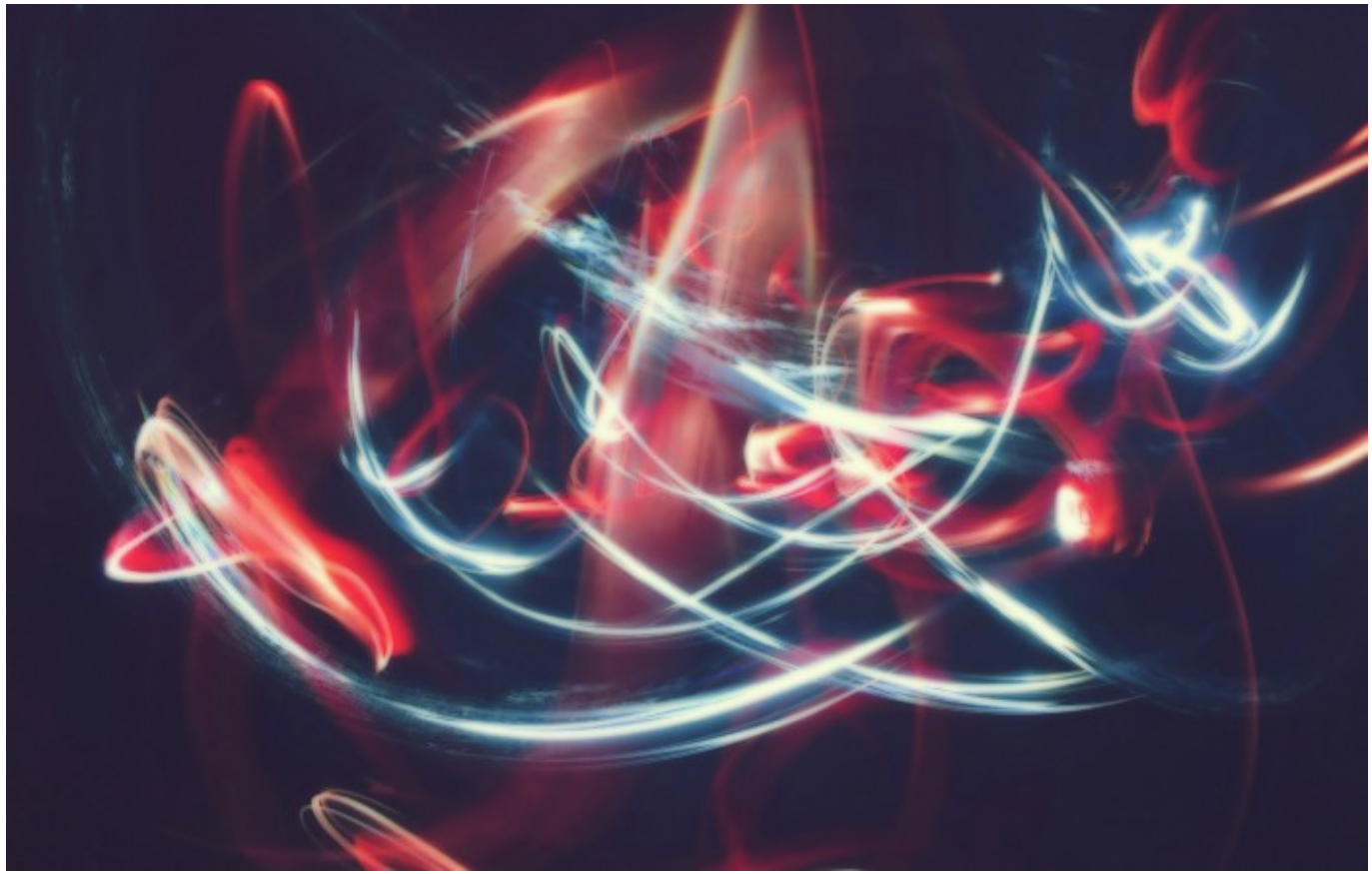


Photo by Ahmad Dirini on Unsplash

Complete Data Analytics Solution Using ETL Pipeline in Python

Diljeet Singh Sethi Follow

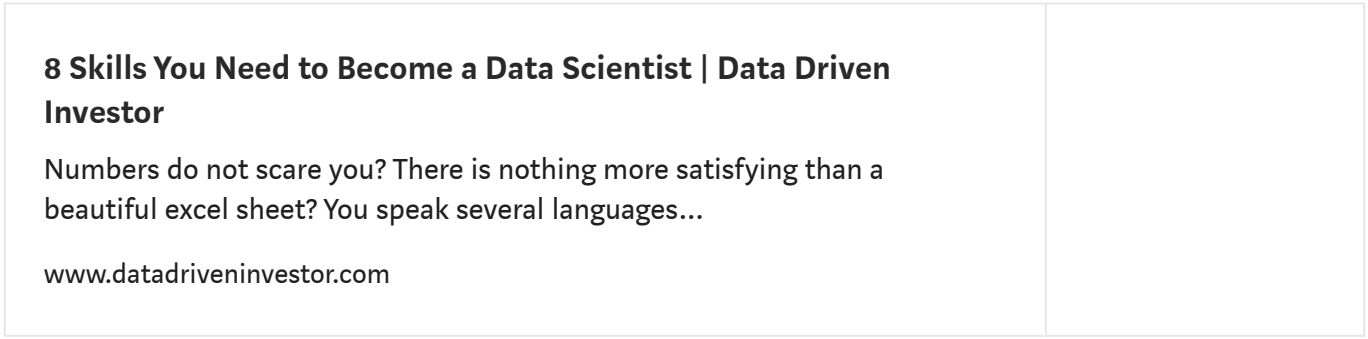
Jul 8, 2019 · 8 min read

> *This blog is about building a configurable and scalable ETL pipeline that addresses to solution of complex Data Analytics projects. Python is used in this blog to build complete ETL pipeline of Data Analytics project.*

We all talk about Data Analytics and Data Science problems and find lots of different solutions. Data Science and Analytics has already proved its necessity in the world and we all know that the future isn't going forward without it.

But what a lot of developers or non-developers community still struggle with is building a nice configurable, scalable and a modular code pipeline,

when they are trying to integrate their Data Analytics solution with their entire project's architecture.



Here in this blog, I will be walking you through a series of steps that will help you understand better about how to provide an end to end solution to your data analysis solution when building an ETL pipe.

So let's start with a simple question, that is, ***What is ETL and how it can help us with Data Analysis solutions ???***

Answer to the first part of the question is quite simple, ETL stands for Extract, Transform and Load. As the name suggests, it's a process of extracting data from one or multiple data sources, then, transforming the data as per your business requirements and finally loading the data into data warehouse. To understand basic of ETL in Data Analytics, refer to this blog.

Understanding Extract, Transform and Load (ETL) and its necessity in Data Analytics world with...

Anyone who is into Data Analytics, be it a programmer, business analyst or database developer, has been developing ETL...

medium.com

Here, in this blog we are more interested in building a solution which addresses to complex Data Analytics project where multiple Data Source like API's, Databases or CSV or JSON files etc are required, to handle this much Data Sources we also need to write a lot of code for Transformation part of ETL pipeline. And yes we can have a requirement for multiple data loading resources as well.

So in my experience, at an architecture level, the following concepts should always be kept in mind when building an ETL pipeline.

> ***Architecture Design Concepts***

Configurability: By definition, it means to design or adapt to form a specific configuration or for some specific purpose. But that isn't much clear. You can think of it as an extra JSON, XML or name-value pairs file in

your code that contains information about databases, API's, CSV files, etc. For example, if I have multiple data source to use in code, it's better if I create a JSON file that will keep track of all the properties of these data sources instead of hardcoding it again and again in my code at the time of using it. *But what's the benefit of doing it? A decrease* in code size, as we don't need to mention it again in our code. It simplifies the code for future flexibility and maintainability, as if we need to change our API key or database hostname, then it can be done relatively easy and fast, just by updating it in the config file.

Modularity or Loosely-Coupled: It means dividing your code into independent components whenever possible. For example, let's assume that we are using Oracle Database for data storage purpose. So if we code a separate class for Oracle Database in our code, which consist of generic methods for Oracle Connection, data Reading, Insertion, Updation, and Deletion, then we can use this independent class in any of our project which makes use of Oracle database. The idea is that internal details of individual modules should be hidden behind a public interface, making each module easier to understand, test and refactor independently of others.

Scalability: It means that Code Architecture is able to handle new requirements without much change in the code base. In our case, this is of

utmost importance, since in ETL, there could be requirements for new transformations. So we need to build our code base in such a way that adding new code logic or features are possible in the future without much alteration with the current code base. We can take help of OOP's concept here, this helps with code Modularity as well.

> ***Data Analytics example with ETL in Python***

Prerequisite for Code:

- Python Basics
- Classes and Objects
- Pandas and Numpy
- API's (request module)
- JSON and CSV files
- MongoDB (pymongo module)

Let's dig into coding our pipeline and figure out how all these concepts are applied in code. I will be creating a project in which we use Pollution data, Economy data and Cryptocurrency data. Let's assume that we want to do

some data analysis on these data sets and then load it into MongoDB database for critical business decision making or whatsoever.

Data Sources link:

API : These API's will return data in JSON format.

- Pollution Data: "https://api.openaq.org/v1/latest?country=IN&limit=10000" .
- Economy Data: "https://api.data.gov.in/resource/07d49df4-233f-4898-92db-e6855d4dd94c?api-key=579b464db66ec23bdd000001cdd3946e44ce4aad7209ff7b23ac571b&format=json&offset=0&limit=100"

CSV Data about Crypto Currencies: https://raw.githubusercontent.com/diljeet1994/Python_Tutorials/master/Projects/Advanced%20ETL/crypto-markets.csv

I have taken different types of data here since in real projects there is a possibility of creating multiple transformations based on different kind of data and its sources.

Configuration

To handle it, we will create a JSON config file, where we will mention all these data sources. We will create 'API' and 'CSV' as different key in JSON file and list down data sources under both the categories. Here is a JSON file.

```
{
        "data_sources": {
                "api": {
                        "Pollution": "https://api.openaq.org/v1/latest?country=IN&limit=10000",
                        "Economy": "https://api.data.gov.in/resource/07d49df4-233f-4898-92db-e68
                },
                "csv": {
                        "CryptoMarkets": "crypto-markets.csv"
                }
        }
}
```

data_config.json hosted with ♡ by GitHub view raw

Now in future, if we have another data source, let's assume MongoDB, we can add its properties easily in JSON file, take a look at the code below:

```
{
```

```
        "data_sources": {
                "api": {
                        "Pollution": "https://api.openaq.org/v1/latest?country=IN&limit=10000",
                        "Economy": "https://api.data.gov.in/resource/07d49df4-233f-4898-92db-e68
                },
                "csv": {
                        "Cryptomarkets": "crypto-markets.csv"
                },
                "mongoDB": {
                        "username": "username",
                        "password": "encrypted_password",
                        "host": "host name"
                }
        }
}
```

data_config1.json hosted with ♡ by GitHub view raw

Extraction

Since our data sources are set and we have a config file in place, we can start with the coding of Extract part of ETL pipeline. It's best to create a class in python that will handle different data sources for extraction purpose. Also, by coding a class, we are following OOP's methodology of programming and keeping our code **modular or loosely coupled**. Since we are using APIS and CSV file only as our data source, so we will create two

generic functions that will handle API data and CSV data respectively. take a look at the code below:

```python
import pandas as pd
import requests  # this package is used for fetching data from API
import json

class Extract:

    def __init__(self):
       # loading our json file here to use it across different class methods
        self.data_sources = json.load(open('data_config.json'))
        self.api = self.data_sources['data_sources']['api']
        self.csv_path = self.data_sources['data_sources']['csv']


    def getAPISData(self, api_name):
        # since we have multiple API's (Pollution and Economy Data),
        # so we can get apt API link by passing in its name in function argument.

        api_url = self.api[api_name]
        response = requests.get(api_url)
        # response.json() will convert json data into Python dictionary.
        return response.json()


    def getCSVData(self, csv_name):
        # since we can use multiple CSV data files in future,
        # so will pass csv name as an argument to fetch the desired CSV data.
        df = pd.read_csv(self.csv_path[csv_name])
        return df
```

```
28          return df
29
```

extract.py hosted with ♡ by GitHub view raw

We talked about **scalability** as well earlier. If you take a look at the above code again, you will see we can add more generic methods such as MongoDB or Oracle Database to handle them for data extraction. Since methods are generic and more generic methods can be easily added, so we can easily reuse this code in any project later on.

Loading

Let's create another module for Loading purpose. I will be creating a class to handle MongoDB database for data loading purpose in our ETL pipeline. The code will be again based on concepts of Modularity and Scalability. Take a look at the code below:

```
1  from pymongo import MongoClient
2  import pandas as pd
3
4
5  class MongoDB:
6
7      # Initilize the common usable variables in below function:
8      def __init__(self, user, password, host, db_name ,port='27017', authSource='admin'):
```

```
        self.user = user
        self.password = password
        self.host = host
        self.port = port
        self.db_name = db_name
        self.authSource = authSource
        self.uri = 'mongodb://' + self.user + ':' + self.password + '@'+ self.host + ':' + self.
        try:
            self.client = MongoClient(self.uri)
            self.db = self.client[self.db_name]
            print('MongoDB Connection Successful. CHEERS!!!')
        except Exception as e:
            print('Connection Unsuccessful!! ERROR!!')
            print(e)

    # Function to insert data in DB, could handle Python dictionary and Pandas dataframes
    def insert_into_db(self, data, collection):
        if isinstance(data, pd.DataFrame):
            try:
                self.db[collection].insert_many(data.to_dict('records'))
                print('Data Inserted Successfully')
            except Exception as e:
                print('OOPS!! Some ERROR Occurred')
                print(e)
        else:
            try:
                self.db[collection].insert_many(data)
                print('Data Inserted Successfully')
            except Exception as e:
                print('OOPS!! Some ERROR Occurred')
                print(e)

```

```
41      def read_from_db(self, collection):
42          try:
43              data = pd.DataFrame(list(self.db[collection].find()))
44              print('Data Fetched Successfully')
45              return data
46          except Exception as e:
47              print('OOPS!! Some ERROR Occurred')
48              print(e)
49
50
```

load.py hosted with ♡ by GitHub view raw

Here, you can see that MongoDB connection properties are being set inside MongoDB Class initializer (this function __init__()), keeping in mind that we can have multiple MongoDb instances in use. So whenever we create the object of this class, we will initialize it with that particular MongoDB instance properties that we want to use for reading or writing purpose.

Method for insertion and reading from MongoDb are added in the code above, similarly, you can add generic methods for Updation and Deletion as well. Try it out yourself and play around with the code.

Transformation

We can start with coding Transformation class. Since transformations are based on business requirements so keeping modularity in check is very tough here, but, we will make our class scalable by again using OOP's concept.

So far we have to take care of 3 transformations, namely, Pollution Data, Economy Data, and Crypto Currencies Data. Since transformation logic is different for different data sources, so we will create different class methods for each transformation.

For the sake of simplicity, try to focus on class structure and understand the view behind designing it. I am not saying that this is the only way to code it but definitely it is one way and does let me know in comments if you have better suggestions.

Also if you have any doubt understanding the code logic or data source, kindly ask it out in comments section.

Okay, first take a look at the code below and then I will try to explain it.

```
1  from DataSources import Extract
2  from DataLoad import MongoDB
3  import urllib
```

```
import pandas as pd
import numpy as np

class Transformation:

    def __init__(self, dataSource, dataSet):

        # creating Extract class object here, to fetch data using its generic methods for APIS a
        extractObj = Extract()

        if dataSource == 'api':
            self.data = extractObj.getAPISData(dataSet)
            funcName = dataSource+dataSet

            # getattr function takes in function name of class and calls it.
            getattr(self, funcName)()
        elif dataSource == 'csv':
            self.data = extractObj.getCSVData(dataSet)
            funcName = dataSource+dataSet
            getattr(self, funcName)()
        else:
            print('Unkown Data Source!!! Please try again...')

    # Economy Data Transformation
    def apiEconomy(self):
        gdp_india = {}
        for record in self.data['records']:
            gdp={}

            # taking out yearly GDP value from records
            gdp['GDP_in_rs_cr'] = int(record['gross_domestic_product_in_rs_cr_at_2004_05_prices'
            gdp_india[record['financial_year']] = gdp
```

```
            gdp_india_yrs = list(gdp_india)

        for i in range(len(gdp_india_yrs)):
            if i == 0:
                pass
            else:
                key = 'GDP_Growth_' + gdp_india_yrs[i]
                # calculating GDP growth on yearly basis
                gdp_india[gdp_india_yrs[i]][key] = round(((gdp_india[gdp_india_yrs[i]]['GDP_in_r

        # connection to mongo db
        mongoDB_obj = MongoDB(urllib.parse.quote_plus('root'), urllib.parse.quote_plus('password
        # Insert Data into MongoDB
        mongoDB_obj.insert_into_db(gdp_india, 'India_GDP')

    # Pollution Data Transformation
    def apiPollution(self):
        air_data = self.data['results']

        # Converting nested data into linear structure
        air_list = []
        for data in air_data:
            for measurement in data['measurements']:
                air_dict = {}
                air_dict['city'] = data['city']
                air_dict['country'] = data['country']
                air_dict['parameter'] = measurement['parameter']
                air_dict['value'] = measurement['value']
                air_dict['unit'] = measurement['unit']
                air_list.append(air_dict)

        # Convert list of dict into pandas df
        df = pd.DataFrame(air_list, columns=air_dict.keys())
```

```
69
70          # connection to mongo db
71          mongoDB_obj = MongoDB(urllib.parse.quote_plus('root'), urllib.parse.quote_plus('password
72          # Insert Data into MongoDB
73          mongoDB_obj.insert_into_db(df, 'Air_Quality_India')
74
75      # Crypto Market Data Transformation
76      def csvCryptoMarkets(self):
77          assetsCode = ['BTC','ETH','XRP','LTC']
78
79          # coverting open, close, high and low price of crypto currencies into GBP values since c
80          # if currency belong to this list ['BTC','ETH','XRP','LTC']
81          self.csv_df['open'] = self.csv_df[['open', 'asset']].apply(lambda x: (float(x[0]) * 0.75
82          self.csv_df['close'] = self.csv_df[['close', 'asset']].apply(lambda x: (float(x[0]) * 0.
83          self.csv_df['high'] = self.csv_df[['high', 'asset']].apply(lambda x: (float(x[0]) * 0.75
84          self.csv_df['low'] = self.csv_df[['low', 'asset']].apply(lambda x: (float(x[0]) * 0.75)
85
86          # dropping rows with null values by asset column
87          self.csv_df.dropna(inplace=True)
88
89          # saving new csv file
90          self.csv_df.to_csv('crypto-market-GBP.csv')
```

transformation.py hosted with ♡ by GitHub view raw

Code section looks big, but no worries, the explanation is simpler. So let's start with initializer, as soon as we make the object of Transformation class with dataSource and dataSet as a parameter to object, its initializer will be invoked with these parameters and inside initializer, Extract class object

will be created based on parameters passed so that we fetch the desired data. Again based on parameters passed (datasource and dataset) when we created Transformation Class object, Extract class methods will be called and following it transformation class method will be called, so it's kind of automated based on the parameters we are passing to transformation class's object.

Now, transformation class's 3 methods are as follow:

- apiEconomy(): It takes economy data and calculates GDP growth on a yearly basis.
- apiPollution(): this functions simply read the nested dictionary data, takes out relevant data and dump it into MongoDB.
- csvCryptomarkets(): this function reads data from a CSV file and converts the cryptocurrencies price into Great Britain Pound(GBP) and dumps into another CSV.

We can easily add new functions based on new transformations requirement and manage its data source in the config file and Extract class.

Also, if we want to add another resource for Loading our data, such as Oracle Database, we can simply create a new module for Oracle Class as we did for MongoDB.

Automating the ETL pipeline

The only thing that is remaining is, how to automate this pipeline so that even without human intervention, it runs once every day.

For that we can create another file, let's name it main.py, in this file we will use Transformation class object and then run all of its methods one by one by making use of the loop. Take a look at the code snippet below.

```
from Transformation import Transformation
import json

class Engine:

    def __init__(self, dataSource, dataSet):
        trans_obj = Transformation(dataSource, dataSet)


if __name__ == '__main__':

    etl_data = json.load(open('data_config.json'))
    for dataSource, dataSet in etl_data['data_sources'].items():
```

```
14          print(dataSource)
15          for data in dataSet:
16              print(data)
17              main_obj = Engine(dataSource, data)
18
19
```

main.py hosted with ♡ by GitHub view raw

Since transformation class initializer expects dataSource and dataSet as parameter, so in our code above we are reading about data sources from data_config.json file and passing the data source name and its value to transformation class and then transformation class Initializer will call the class methods on its own after receiving Data source and Data Set as an argument, as explained above.

To run this ETL pipeline daily, set a cron job if you are on linux server. You can also make use of Python Scheduler but that's a separate topic, so won't explaining it here.

Here is GitHub url to get the jupyter notebooks for the whole project. https://github.com/diljeet1994/Python_Tutorials/tree/master/Projects/Advanced%20ETL