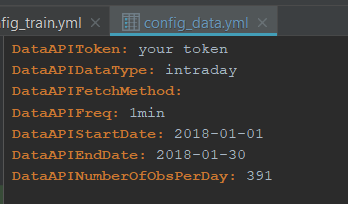
**代码讲解**

**2.1 Config**



The execution is governed by the **config** (dictionary). This component allows us to encapsulate a lot of executions and tidy up the code. It can also be used as a carrier of additional parameters.

For instance, in the previous section, the instantiation of **API.Tiingo** takes the config as an input set it to an attribute. When it calls the underlying functions, the input parameters such as start date, end date, token, no. of sample per day and data frequency will be extracted from the config.



config\_data.yml for data fetching

Currently only a single config is implemented. Ideally, we should implement multiple configs for different components.

Using the **PyYAML**package the code can recognize the fields in ***.yaml*/**.***yml***file and convert the format *automatically*:

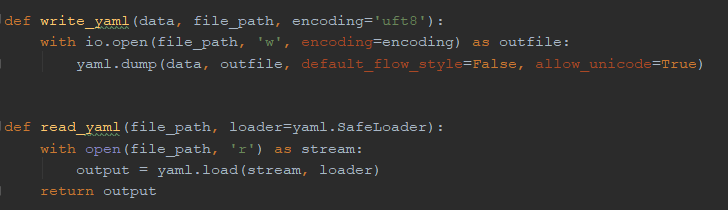
**- Empty field**: loaded into **None  
- True/False**: loaded into Boolean field **True** or **False  
- 1.0**: loaded into float **1.0  
- 1**: loaded into integer **1  
- string**: loaded into **‘string’  
- [1, 2, 3]**: loaded into list **[1, 2, 3]  
- 2018–01–01**: loaded into **datetime.date(2018, 1, 30)  
-**Finally, if we put this into the yaml file:

**Folder:  
 Folder A:** Math Notes  
 **Folder B:** [Memo, Magazines]

the package can recognize the indentation and load it into a dictionary:

{'Folder A': 'Math Notes', 'Folder B': ['Memo', 'Magazines']}

Check the **UTIL/FileIO.py** for the reading and writing functions:



**2.2 Data API**

For this we have already covered the main detail so I am gonna skip this. If you would like to add another API I would suggest you to simply make another class, with the same interface as **fetch**in the class **Tiingo**.

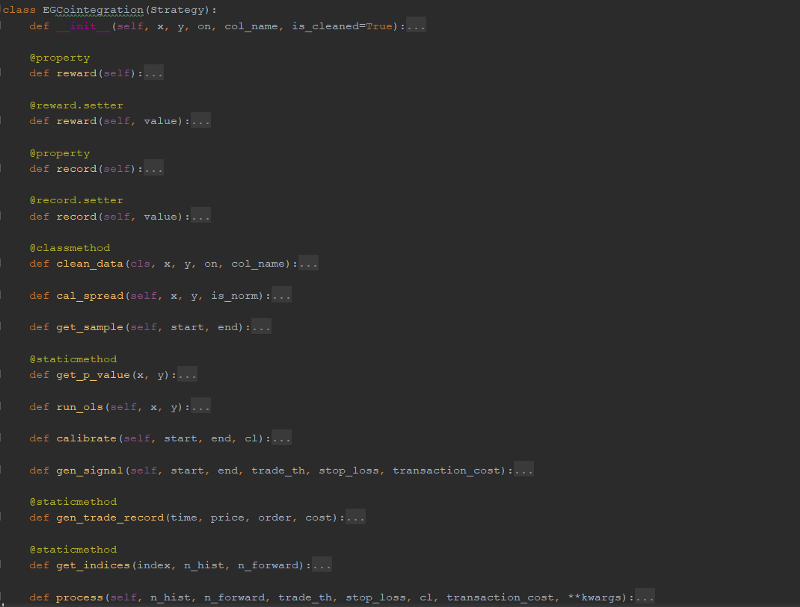
**2.3 Strategy**



In **./STRATEGY**each module contains a strategy category, each strategy should be represented by one class. The class is inherited from an [abstract base class](https://docs.python.org/3/library/abc.html) which requires it to implement the following:

1. **process()**: called by the machine learning script during training or testing
2. **reward**: properties that define the RL reward (i.e. trade profit)
3. **record**: any other attributes to be stored during the training

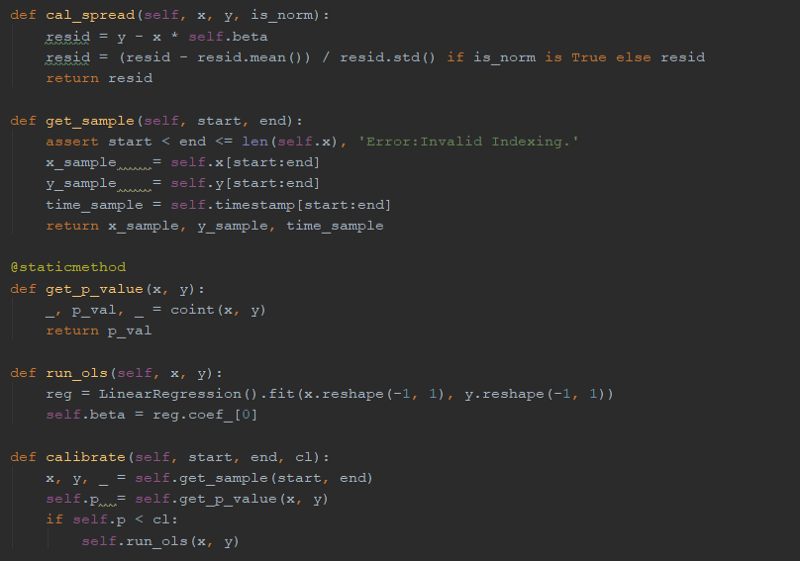
Inside the package we can find a strategy class **EGCointegration** which takes price data **x** and **y** and other parameters during the instantiation. When the underlying functions need a sample data set, they will call the **get\_sample**function to perform the sampling from its data attributes.



EGCointegration class in ./Strategy/Cointegration.py

During the training phase, in each iteration we will need to calibrate the p-value and coefficients to decide whether and how a pair trading should be triggered. These executions are embedded in the same class. when the **process** is called, the object will automatically perform the sampling from its data attributes and run the calibration. Based on the calibrated result the function will get a reward and record and set them to the corresponding attributes.

See more about cointegration and its testing in **Part 3**.



Key functions for calibration in EGCointegration

**2.4 Basic Building Blocks, Processors, and ML Algorithms**



These components are highly integrated and governed not only by the config but also the tailor-made agent which control the whole ML process which is highly automated. Many ML algorithms were hard-coded. That means if the logic needs to be fine tuned, the code has to be amended which is a bit inconvenient. Here, although the design is a bit complicated, if you can understand the style you will be able to expand it in any way you want.

Recently, Google has released an open-source library for reinforcement learning (RL) called [TF-Agents](https://github.com/tensorflow/agents). Feel free to check [this](https://www.youtube.com/watch?v=-TTziY7EmUA) out. Some concepts are similar, but the main focus of our code is on the automation so you may use that as a foundation if you would like to build a new one.

**2.4.1 Basic Building Blocks**

* **Agent**



Agent class in Basic.py

It is the main body that runs and control the processes in ML. In RL, it has another layer of implication: in general it is the component that receives the states of the environment and makes decision on what action to take accordingly. The **Agent** class is meant to be inherited by the machine learning class. It should be initiated with a **Network** object and a **config** dictionary. Major functions include:

- **docking**: attach the Network input and output layers  
- **assign\_network**: assign new Network to the Agent object and connect   
- **set\_session**: set TensorFlow   
- **get\_counter**: extract the parameters from config and get a dictionary of **StepCounter**objects for looping or increments such as varying probability  
- **save\_model / restore\_model**: save and restore model in / from ***.ckpt*** file  
- **process**: abstract method to be implemented for training or testing

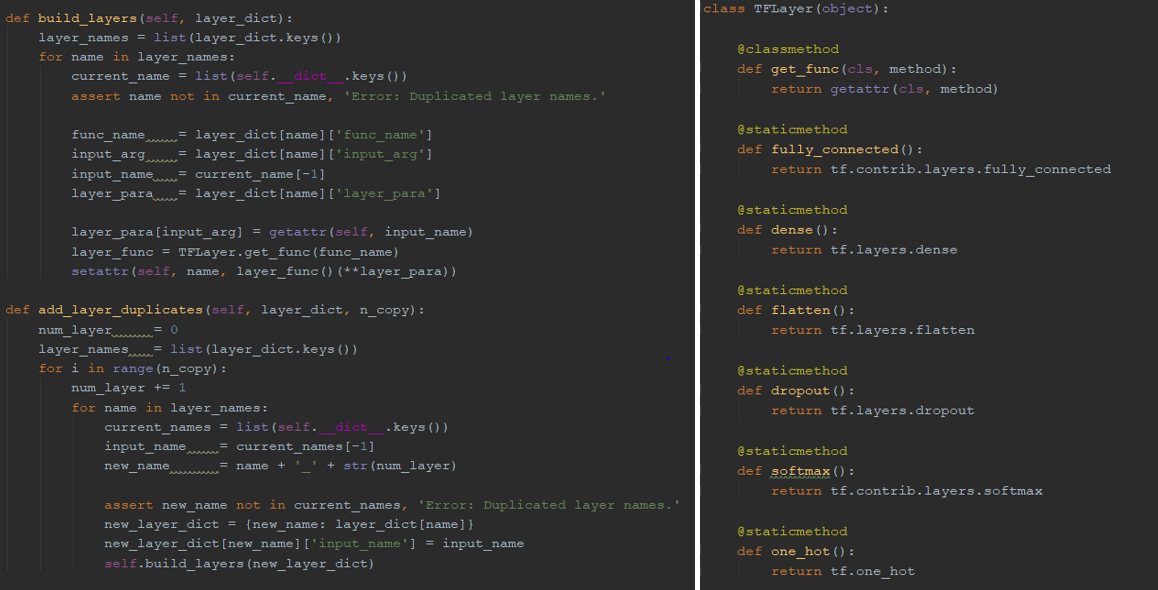
* **Network**

A typical way of building a TensorFlow neural network is something like this inside which the layers and the parameters in each of them are hard-coded:

input\_layer = tf.reshape(x, [-1, 28, 28, 1])  
conv1 = tf.layers.conv2d(inputs=input\_layer,  
 filters=32,  
 kernel\_size=[5, 5],  
 padding="same",  
 activation=tf.nn.relu)  
pool1 = tf.layers.max\_pooling2d(inputs=conv1, pool\_size=[2, 2], strides=2)  
conv2 = tf.layers.conv2d(inputs=pool1,  
...

Alternatively we could also build a function that repeats the above process, forfeiting the flexibility in setting the layer arguments.

If you want to build a ML system or something with GUI with flexibility in customizing the detail for each layer (i.e. layer type, layer inputs, layer arguments) while preserving the automaticity, here comes a suggestion:



Network and TFLayer in Basics.py

The two functions on the left are under the class **Network**.

1. **build\_layers**: it takes a dictionary **layer\_dict** as an input and construct the network by sequentially adding layers selected from the **TFLayer** class as shown on the right hand side. As long as for each layer the parameters are properly defined, this function can be called recursively to add layers on top of the existing final layer in the current network. Every layer is set to the attribute of the **Network**object so their name must be unique.
2. **add\_layer\_duplicates**: similar to **build\_layers**, it takes a **layer\_dict** as an input, and require an input of **n\_copy** which specify how many copies of the layer(s) prescribed by the **layer\_dict** should be added on top of the existing network. New names will be created for the duplicated layers by concatenating the layer name and the number of that layer among the copies.

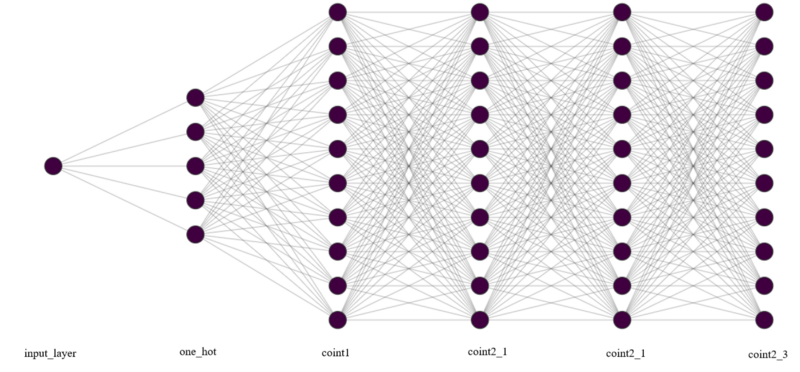
For example:



The steps to create a network:

1. **Initiate an Network object**. This has to be instantiated by the first input layer which is the tf.placeholder in this example.
2. **Build the network based on layer\_dict1**. It specifies 2 layers: an **‘one\_hot’** layer which is actually tf.one\_hot with 5 outputs, and a**‘coint1’**layer which is tf.contrib.layers.fully\_connected with 10 outputs. The input arguments of the tf.contrib.layers.fully\_connected are defined by the key **‘layer\_para’**.
3. **Expand the network by adding copies of layer prescribed by layer\_dict2**. The layer **‘coint2’** with 10 outputs is added to the current network for 3 times.

Therefore, the Network object **N** now should have 6 attributes in total. Each of them is a layer with predefined properties:



Since the construction of the network is based on the layer dictionary, automation comes into ply if the generation of such dictionary is streamlined, and we no longer need to hard code the network every time when we build something new.

* **Space**

Basically it refers to a sample space object. It takes a dictionary of list as an input and create the sample space by making full combinations across list elements. For example, for the following sample space:

space\_dict = {'dice': [1, 2, 3, 4, 5, 6],  
 'coin': ['H', 'T']}  
S = Space.states\_dict

**S** contains **all combinations** of ‘dice’ and ‘coin’, 12 elements in total. It contains the necessary functions that convert the sample from dictionary to a single index, list of indices, or one\_hot array and vice versa that could fit the purpose of adapting different kind of input or output carriers in TensorFlow.

* **StepCounter**

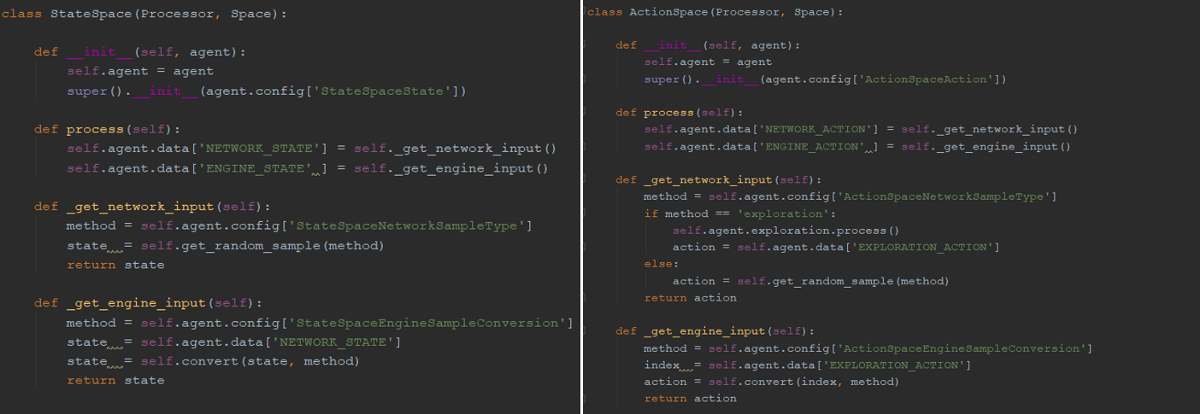
During training, some parameters are incremental such as the current step in for loop, or the learning rate is set to be variable. We may even want to add a buffer before the actual step is triggered (i.e. the learning rate start to drop after 100 loops). Instead of hard coding these in the script, we can have a step counter to perform the above. The counter also incorporates the ability to buffer pre-train steps. For example, the actual counting value starts to change only after 100 buffering steps.

**2.4.2 Processors**

A **Processor** class should take an **Agent**object as an input for initiation. When the **process**is called it will extract relevant parameters from the Agent object, including the attached config dictionary, and attach any output to the data dictionary which is an attribute of the Agent. We can actually create another object to carry these attributes but for simplicity let’s not overload the structure in here.

* **State Space and Action Space**

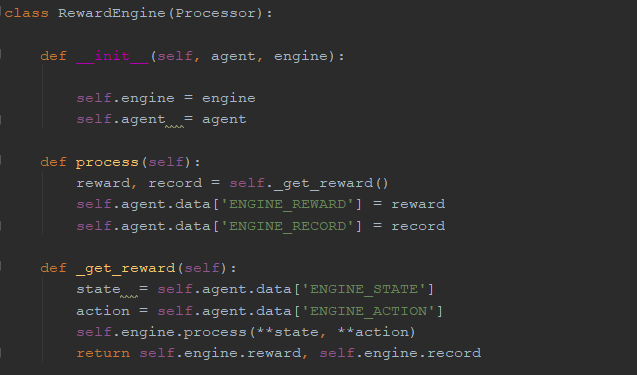
Both of them inherit the parent class **Space** and are used to generate state samples or action samples. Based on the ***method*** specified in config they can output the samples in different forms (i.e. index/one hot/dictionary) or different ways (with/without exploration) serving different purposes such as network training or taken as the input of the ***process*** function in the **Strategy**object.



StateSpace and ActionSpace in PROCESSOR/MachineLearning.py

* **Reward Engine**

It takes an engine object which contain a **process**methods. In our example it will be an **EGCointegration**object.



RewardEngine class

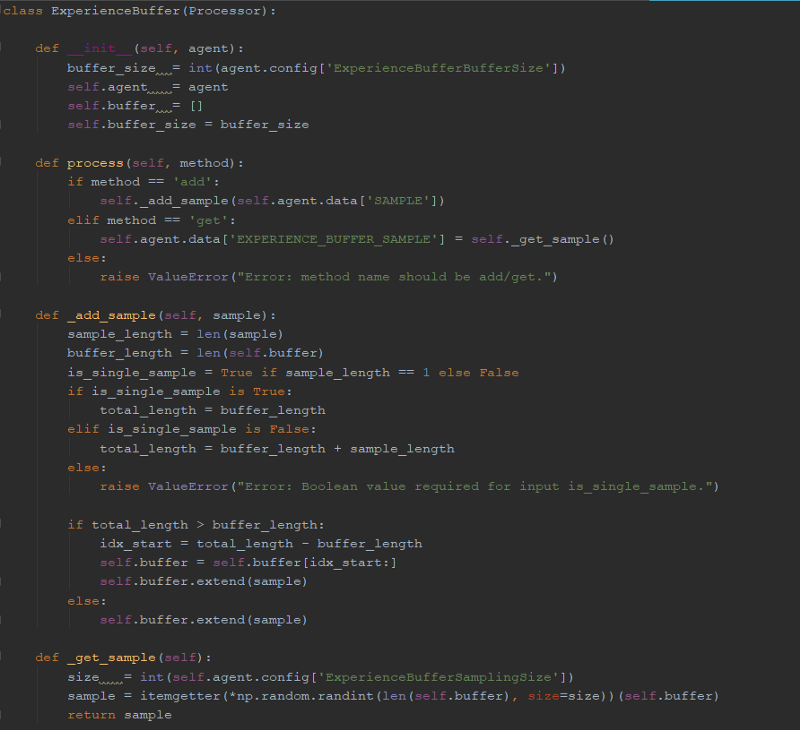
* **Exploration**

The purpose of this object is to explore possible actions. The selected method will return an action index to the data carrier in the Agent object. The exploration is implemented when the **process** function in the ActionSpace is called.



* **Experience Buffer**

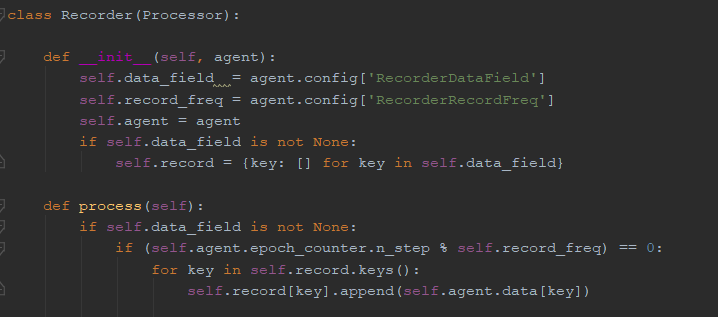
This leverages the **Experience Replay** implementation in this [article](https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df). The purpose is to store the samples and results along the training process, and re-sample from the buffer to allow the agent to re-learn from the history.



* **Recorder**

Last but not least, I created a **Recorder**class which can be used to keep track of the records stored in the data dictionary inside the Agent object. We can select the field we would like it to store by specifying the key names in the ***RecorderDataField*** field in the config file:

**RecorderDataField:** [NETWORK\_ACTION, ENGINE\_REWARD]

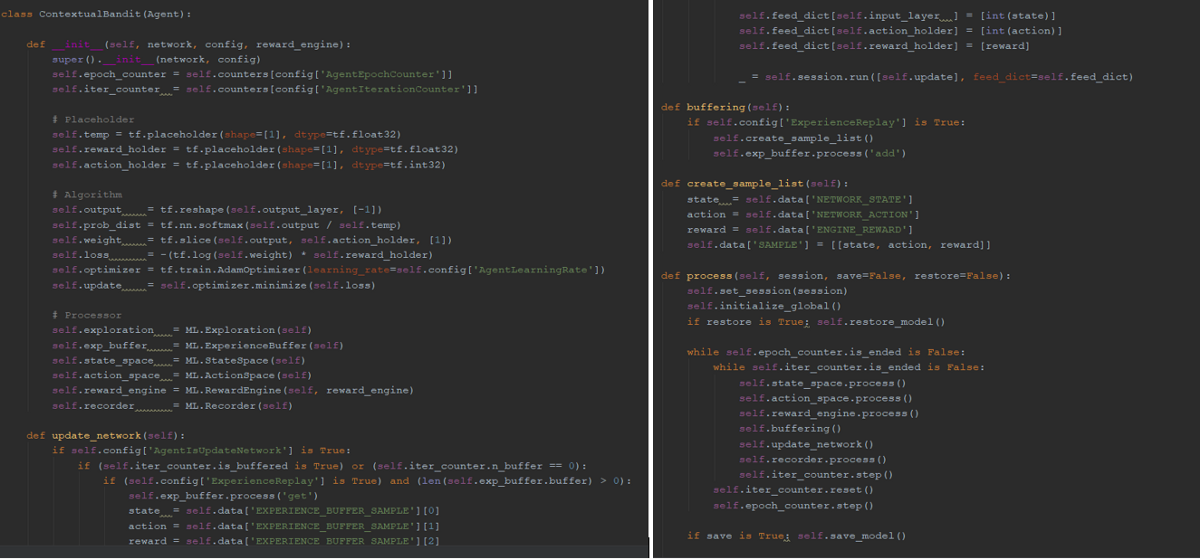


Recorder class

**2.4.3 ML Algorithms**

With the components described above, we can tailor make any class that takes these building blocks and create a running procedure. This is the only part that needs to be customized for different purpose, but still the logic is pretty standardized for similar cases.

For example, in this project I have created a **ContextualBandit**class which can actually perform either N-Armed bandit or contextual bandit running, subject to the number of state. If we would like to run it for N-Armed bandit problem we could just specify a state space with a single fixed state (dummy).



ContextualBandit class in MAIN/Reinforcement.py

1. **\_\_init\_\_**: initiates the object and inherits the parent methods and properties. The TensorFlow machine learning attributes are defined in here as well. After all the processors described above will be instantiated by composition, taking the object itself as an input argument (***agent)***.
2. **update\_network**: extracts the samples from data dictionary and update the TensorFlow layers and network.
3. **buffering**: store the sample in the **ExperienceBuffer** object if specified in the config.
4. **create\_sample\_list**: create samples for experience buffering.
5. **process**: the main procedure that controls the flow of the training or testing. It takes a tf.Session() and perform the looping based on the values in the StepCounter objects initiated by the Agent.