Dal-I/O

An integrated graphical structure for the creation of financial models.

Understanding Graphs

What do I mean by "graphical structure"?

In a graphical structures data is represented as nodes and operations as edges. Think of it as a way to represent many inter-connected transformations and their input and output data.

Why is a graphical structure optimal for financial modeling.

- Modern automated finantial modes retrieve data, clean and dirty, from various sources and through cleaning and itegration are able to join them, further process this product and finally derive insigts. The problem is that as these models utilize more and more data from various sources, created models tend to become confusing for both technical and non technical people. Also, as there is no unified workflow to deal with these, created models tend to become highly inflexible and lacking portability (onto other models or projects.) A graphical architecture offers an intuitive workflow for working with data, where inputs can have a unified translation, data can be constantly checked for validity and otuputs can be used in flexible ways as parts of a bigger system or drive actions.
- Utilizing large ammounts of data can also end up being highly memory-inneficient when data sources are varied and outputs are as simple as a buy/sell command. As in the tensorflow graphical architecture, using these constructs allow for automatic parallelization of models to better use modern hardware. Applications can also be built to fit multiple models, and updated independently from the rest of the system.
- Graphs are easy to interpret visualy, which is useful for understanding the flow of data and interpreting output or bugs. They are also higly flexible, allowing users to modify pieces or generate new connections while keeping an enforcable system of data integrity.
- Perhaps most importantly, these graphs are extremely lightweight and portable, which is key for widespread distribution and access. While every piece can be accessed and tested on-the-go for better ease of development, they are ultimately just pieces of a bigger structure, where data flows continuously and leftover data is dicared automatically, keeping the memory and processing burden at a minimum when dealing with massive datasets.

Base Classes and Consepts

Validator

Validators are the building blocks of data integrity in the graph. As modularity is key, validators ensure that data sourced from a DataDef is what it is mean to be or that errors are targeted to make debugging easier. Validators can have any attribute needed, but functionality is stored in the .validate function, which either passes warning data on or stops execution with an error. These can and should be reused with multiple DataDef instanges.

Node

Node instances represent data. They have a connection to some data input, internal or external, and make requests to this data as well as ensure their integrity. These form the basis for External and DataDef classes.

DataDef

DataDef instances are sources of data and implement mechanisms to ensure the integrity of that data, as input from sources is uncertain.

Descriptions: As you don't necessarily know what source data comes from or the processes it went through, it is important for follow-up tools to know what they are getting. Likewise, as these tools might be used by users unaware of the process the data will go throug here, describing the output is an important way of making models portable.

Validation: In order to hold descriptions true, the data is validated by a chain of functions before returning any actual data, in order to enure that if data is actually returned, it is accurate to its descriptions.

Speed Concerns: As the validator chain might prove to be a computational burden, other options are available to make descriptions possible and lightweight. Validators can be turned on and off as needed with the .blacklist attribute or simply with the .val_off attribute. The .tags attribute is also available to add descriptions to your model without any implied computation. Keep in mind that validation should be disabled with caution, onld do so if you are absolutely certain about the data being returned.

External

Supplicant instances manage connections between your environment and an external source. Class instacnes will often be redundant with existing connection handlers, but at least subclasses will allow for more integrated connection handling and collection, so that you can have a single supplicant object for each external connection.

Transformer

Transformers represent transformations on data. They have one or more sources and one or more outputs, internal or external. Transformers source data from Nodes, transforms them and outputs them.

Pipe

Pipes are a base class that represents any kind of data modification with one internal input and one internal output. All pipes must implement the .transform() method, which takes in the output from sourced data and returns it transformed. The .run() method in turn has a default implementation to actually source the input data from the input node and pass it onto the .transform() method; this default implementation is often changed to modify keyword arguments passed onto the source node and the .transform() call.

PipeLine

As Pipe instances implement a normally small operation and have only one imput and one output, you are able to join them together, thorugh the ___add___() internal method to create a sequence of transformations linked one after the other. These simply pass the output of one Pipe instance's .transform() method as the input to another, so be carefull with data integrity here.

Translator

Translators are the root of all data that feeds your graph. Objects of this class connect with some external source, imports raw data, then "translates" it into a format that can be used universaly through the model.

Configuration: Sources often require additional ids, secrets or paths in order to access their data. The .config attribute aims to summarise all key configuration details and data needed to access a resource. Additional functions can be added as needed to facilitate one-time connection needs.

Factories: Sources, typically web APIs, will give users various functionalities with the same base configurations. The .make() method can be implemeted to return subclasses that inherit parent processing and configuration.

Model

Models are a lot like transformers as they take in inputs and has a single output. Models do differ from transformers as they can take in multiple inputs and be much more flexible with additional methods for different strategies or for small data storage. Also, keep in mind models do not have a ___call___ method inherited or a single function that transforms its inputs. Models are supposed to perform more intricate operations, beyond a simple transformation.

Applications

While Models are normally the last stage of a model, it still has a single output which might have limited value in itself. Interpreters are tools used for the interpretation of some DataDef, which are not constrained by those output by models, but often are. These can have a broad range of applications, from graphing to trading. The main functionality is in the .execute() method, which gets input data and interprets it as needed.

Key Concepts, Differences and Philosophy

running vs getting

You might have notices that classes that inherit from have .run() methods, classes that inherit from have .get() methods, both of which return some forrm of data. While these two essentally have the same output functionality, they differ in implementation, where .run() methods get data from a source and modifies is while .get() methods get data, also from some source, and validates it. Thus, the idea of a DataOrigin compared to a Pipe becomes clearer.

describing vs tagging

The .tags and .desc attributes might seem to be redundant, as both are used to describe some sort of data passing by them and both can be used to search for nodes in the graph. Firsly, and most importantly, the .desc attribute is common to all DataDef instances that inherit from another DataDef, while the .tag attribute is unique to that node, unless it is also present on the parent DataDef or shared with other DataDefs upon instanciation.

They also do defer in “strictness,” as tags will not be checked for truthfulness, while descriptions will be tested on the data, unless, of course, users turn checking off. Tags are included as a feature to allow more flexible, personalizable descriptions thatdescribe groups or structures within the graph rather than a certain functionality.