Fn Graph

Jul 01, 2020

Contents

CHAPTER 1

Fn Graph

Lightweight function pipelines for python

For more information and live examples look at fn-graph.businessoptics.biz

1.1 Overview

`fn_graph` is trying to solve a number of problems in the python data-science/modelling domain, as well as making it easier to put such models into production.

It aims to:

1. Make moving between the analyst space to production, and back, simpler and less error prone.
2. Make it easy to view the intermediate results of computations to easily diagnose errors.
3. Solve common analyst issues like creating reusable, composable pipelines and caching results.
4. Visualizing models in an intuitive way.

There is an associated visual studio you should check out at https://github.com/BusinessOptics/fn_graph_studio/.

1.2 Documentation

Please find detailed documentation at https://fn-graph.readthedocs.io/

1.3 Installation

```
pip install fn_graph
```

You will need to have graphviz and the development packages installed. On ubuntu you can install these with:

```
sudo apt-get install graphviz graphviz-dev
```

Otherwise see the pygraphviz documentation.

To run all the examples install

```
pip install fn_graph[examples]
```

1.4 Features

- **Manage complex logic**Manage your data processing, machine learning, domain or financial logic all in one simple unified framework. Make models that are easy to understand at a meaningful level of abstraction.
- **Hassle free moves to production**Take the models your data-scientist and analysts build and move them into your production environment, whether thats a task runner, web-application, or an API. No recoding, no wrapping notebook code in massive and opaque functions. When analysts need to make changes they can easily investigate all the models steps.
- **Lightweight**Fn Graph is extremely minimal. Develop your model as plain python functions and it will connect everything together. There is no complex object model to learn or heavy weight framework code to manage.
- **Visual model explorer**Easily navigate and investigate your models with the visual fn_graph_studio. Share knowledge amongst your team and with all stakeholders. Quickly isolate interesting results or problematic errors. Visually display your results with any popular plotting libraries.
- **Work with or without notebooks**Use fn_graph as a complement to your notebooks, or use it with your standard development tools, or both.
- **Works with whatever libraries you use**fn_graph makes no assumptions about what libraries you use. Use your favorite machine learning libraries like, scikit-learn, PyTorch. Prepare your data with data with Pandas or Numpy. Crunch big data with PySpark or Beam. Plot results with matplotlib, seaborn or Plotly. Use statistical routines from Scipy or your favourite financial libraries. Or just use plain old Python, it's up to you.
- **Useful modelling support tools**Integrated and intelligent caching improves modelling development iteration time, a simple profiler works at a level that's meaningful to your model. ** *Easily compose and reuse models**The composable pipelines allow for easy model reuse, as well as building up models from simpler sub-models. Easily collaborate in teams to build models to any level of complexity, while keeping the individual components easy to understand and well encapsulated.
- **It's just Python functions**It's just plain Python! Use all your existing knowledge, everything will work as expected. Integrate with any existing python codebases. Use it with any other framework, there are no restrictions.

1.5 Similar projects

An incomplete comparison to some other libraries, highlighting the differences:

Dask

Dask is a light-weight parallel computing library. Importantly it has a Pandas compliant interface. You may want to use Dask inside FnGraph.

Airflow

Airflow is a task manager. It is used to run a series of generally large tasks in an order that meets their dependencies, potentially over multiple machines. It has a whole scheduling and management apparatus around it. Fn Graph is not

trying to do this. Fn Graph is about making complex logic more manageable, and easier to move between development and production. You may well want to use Fn Graph inside your airflow tasks.

Luigi

> Luigi is a Python module that helps you build complex pipelines of batch jobs. It handles dependency resolution, workflow management, visualization etc. It also comes with Hadoop support built in.

Luigi is about big batch jobs, and managing the distribution and scheduling of them. In the same way that airflow works ate a higher level to FnGraph, so does luigi.

d6tflow

d6tflow is similar to FnGraph. It is based on Luigi. The primary difference is the way the function graphs are composed. d6tflow graphs can be very difficult to reuse (but do have some greater flexibility). It also allows for parallel execution. FnGraph is trying to make very complex pipelines or very complex models easier to mange, build, and productionise.

CHAPTER 2

Usage

2.1 Building up logic

That principle idea behind Fn Graph is to use the names of a functions arguments to find that functions dependencies, and hence wire up the graph.

There are multiple methods to add functions to the graph, all of them use the underlying `update` function. In the most direct form update takes keyword arguments, the keyword defines the name of the function in the graph. For example:

```
from fn_graph import Composer

def get_a():
    return 5

def get_b(a):
    return a * 5

def get_c(a, b):
    return a * b

composer = Composer().update(a=get_a, b=get_b, c=get_c)
```

Alternatively, some may prefer the more terse version where the function name itself is used as the name within the graph, for example:

```
from fn_graph import Composer

def a():
    return 5

def b(a):
    return a * 5

```

(continues on next page)

(continued from previous page)

```
def c(a, b):
    return a * b

composer = Composer().update(a=a, b=b, c=c)
```

The issue with this is it leads to name shadowing, which some people don't like, and many linters will (rightly) complain about. An alternative that is reasonably terse and does not have the name shadowing problem is to use the prefix or suffix stripping versions of update.

```
from fn_graph import Composer

def get_a():
    return 5

def get_b(a):
    return a * 5

def get_c(a, b):
    return a * b

composer = Composer().update_without_prefix("get_", get_a, get_b, get_c)
```

Often you have static inputs into a graph, parameters. It is more convenient ot treat these differently rather than creating functions that just return the values, and use the `update_parameters` method.

```
from fn_graph import Composer

def get_b(a):
    return a * 5

def get_c(a, b):
    return a * b

composer = (
    Composer()
    .update_without_prefix("get_", get_b, get_c)
    .update_parameters(a=5)
)
```

All update methods return a nw composer, and as such can be safely chained together. You can also update a composer with all the functions of another composer using the `update_from` method.

```
composer_a = ...

composer_b = ...

composer_c = composer_b.update_from(composer_a)
```

2.2 Visualisation

You can see the function graph using the `graphviz` method. In a notebook environment this will be rendered directly. In other environment you may want to use the view method.

```
# In a notebbook
composer.graphviz()

#In other environments
composer.graphviz().view()
```

2.3 Calculation

The function graph can be calculated using the `calculate` method. It can calculate multiple results at once, and can return all the intermediate results. It returns a dictionary of the results

```
composer.calculate(["a" ,"c"]) // {"a": 5, "c": 125}
composer.calculate(["a" ,"c"], intermediates=True) // {"a": 5, "b": 25, "c": 125}
```

You can als use the call function if you want only a single result.

```
composer.call("c") // 125
```

As an accelerator calling a function is exposed as a method on the composer.

```
composer.c() // 125
```

2.4 Caching

Caching is provided primarily to make development easier. The development cache will cache results to disk, so it persists between sessions. Additionally it stores a hash of the various functions in the composer, and will invalidate the cache when a change is made. This works well as long as functions are pure, but it does not account for changes in things like data files. you can activate this using the `development_cache` method. The method takes a string name which identifies the cache, often you can just use `__name__`, unless the composer is in the `__main__` script.

```
cached_composer = composer.development_cache(__name__)
```

If something has changed that requires the cache to be invalidated you can use the `cache_invalidate` or `cache_clear` methods. `cache_invalidate` takes the names of teh functions you wish to invalidate, it will ensure any follow on functions are invalidated. `cache_clear` will clear the cache.

2.5 Namespaces

When logic gets complex, or similar logic needs to be reused in different spaces in one composer it can useful to use namespaces. Namespaces create a hierarchy of named scopes. that limits what functions arguments resolve to. Namespaces are separated with the double underscore (__). Namespaces are constructed using the `update_namespaces` method. For example:

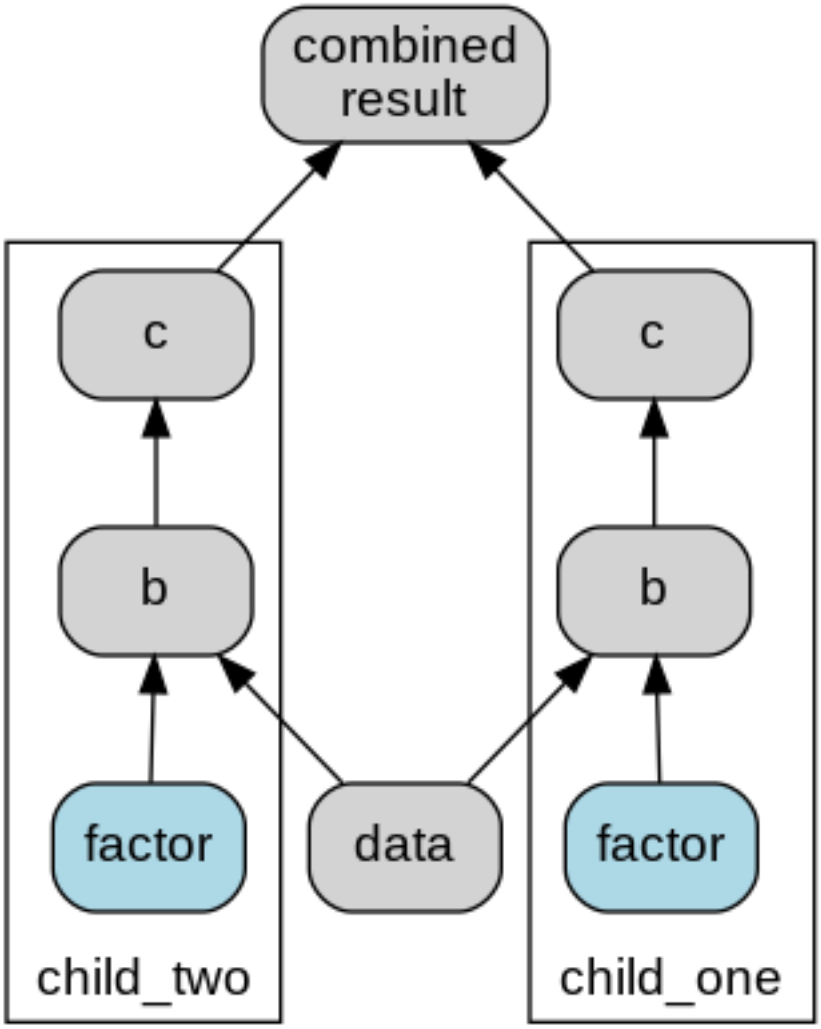
```
from fn_graph import Composer

def data():
    return 5
```

(continues on next page)

(continued from previous page)

```
def b(data, factor):
    return data * factor


def c(b):
    return b


def combined_result(child_one__c, child_two__c):
    pass


child = Composer().update(b, c)
parent = (
    Composer()
    .update_namespaces(child_one=child, child_two=child)
    .update(data, combined_result)
    .update_parameters(child_one__factor=3, child_two__factor=5)
)
```

Then the resulting graph would look like this:



namespaces graphviz

There are couple of things going on here.

1. You can see that function `c` resolves it's arguments within it;s namespace.
2. The `b` functions do not find a `data` function in their own namespace so they look to the parent namespace.
3. The combined_result specifically names it's argunents to pull from the namespaces using the double-underscore (__).
4. The parameters have been set differently in different namespaces (but could have been set the same by putting it in the top level namespace).

These capabilities on their own allow you to construct (and reconstruct) very flexible logic. Sometimes though, given that arguments are resolved just by name it is useful to be able to create a link between one name and another. You can do this using the the `link` method. For example:

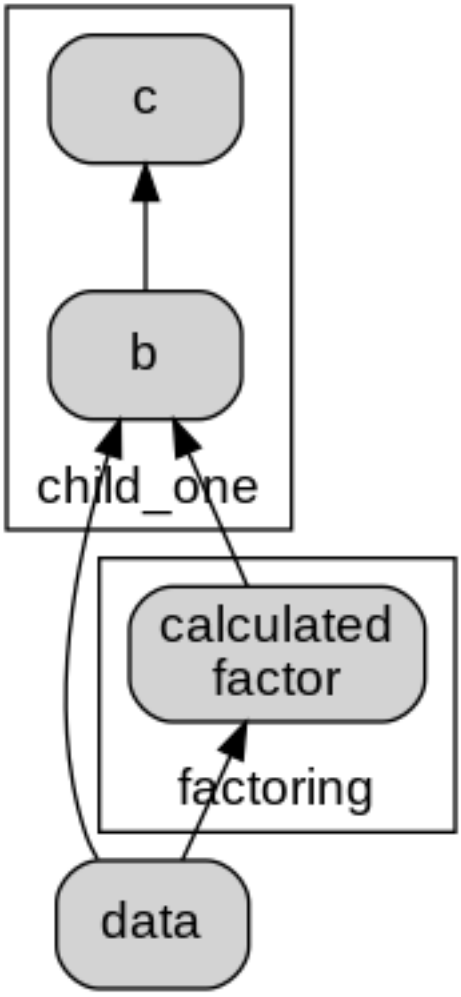
```
def calculated_factor(data):
    return data / 2


factor_calc = Composer()
factoring = Composer().update(calculated_factor)
```

(continues on next page)

(continued from previous page)

```
linked_parent = (
    Composer()
    .update_namespaces(child_one=child, factoring=factoring)
    .update(data)
    .link(child_one__factor="factoring__calculated_factor")
)
```

Which looks like:



linked namespaces graphviz

The link method gives you flexibility to link between different namespaces, and in general reconfigure your logic without having to write new functions.

CHAPTER 3

fn_graph

3.1 fn_graph package

3.1.1 Module contents

class fn_graph.**Composer**(*, *_functions=None*, *_parameters=None*, *_cache=None*, *_tests=None*, *_source_map=None*)

Bases: object

A function composer is responsible for orchestrating the composition and execution of graph of functions.

Pure free functions are added to the composer, the names of the function arguments are used to determine how those functions are wired up into a directed acyclic graph.

ancestor_dag(*outputs*)
A dag of all the ancestors of the given outputs, i.e. the functions that must be calculated to for the given outputs.

cache(*backend=None*) → fn_graph.Composer
Create a new composer with a given cache backend.

By default this is a SimpleCache.

cache_clear()
Clear the cache

cache_graphviz(*outputs=()*, ***kwargs*)
Display a graphviz with the cache invalidated nodes highlighted.

cache_invalidate(**nodes*)
Invalidate the cache for all nodes affected by the given nodes (the descendants).

calculate(*outputs*, *perform_checks=True*, *intermediates=False*, *progress_callback=None*)

call(*output*)
A convenience method to calculate a single output

check(*outputs=None*)
Returns a generator of errors if there are any errors in the function graph.

dag()
Generates the DAG representing the function graph.

Return type a networkx.DiGraph instance with function names as nodes

development_cache(*name*, *cache_dir=None*) → fn_graph.Composer
Create a new composer with a development cache setup

functions()
Dictionary of the functions

get_source(*key*)
Returns the source code that defines this function.

graphviz(*, *hide_parameters=False*, *expand_links=False*, *flatten=False*, *highlight=None*, *filter=None*, *extra_node_styles=None*)
Generates a graphviz.DiGraph that is suitable for display.

This requires graphviz to be installed.

The output can be directly viewed in a Jupyter notebook.

link(***kwargs*)
Create a symlink between an argument name and a function output. This is a convenience method. For example:

f.link(my_unknown_argument="my_real_function")

is the same as

f.update(my_unknown_argument= lambda my_real_function: my_real_function)

parameters()
Dictionary of the parameters of the form {key: (type, value)}

precalculate(*outputs*)
Create a new Composer where the results of the given functions have been pre-calculated.

Parameters **outputs** – list of the names of the functions to pre-calculate

Return type A composer

raw_function(*name*)
Access a raw function in the composer by name. Returns None if not found.

run_tests()
Run all the composer tests.

Returns A generator of ComposerTestResults(name, passed, exception).

set_source_map(*source_map*)
Source maps allow you to override the code returned by get source.

This is rarely used, and only in esoteric circumstances.

subgraph(*function_names*)
Given a collection of function names this will create a new composer that only consists of those nodes.

update(**args*, ***kwargs*) → fn_graph.Composer
Add functions to the composer.

Parameters **args** – Positional arguments use the __name__ of the function as the reference

in the graph. kwargs: Keyword arguments use the key as the name of the function in the graph.

Returns A new composer with the functions added.

update_from(**composers*) → fn_graph.Composer
Create a new composer with all the functions from this composer as well as the the passed composers.

Parameters **composers** – The composers to take functions from

Returns A new Composer with all the input composers functions added.

update_namespaces(***namespaces*) → fn_graph.Composer
Given a group of keyword named composers, create a series of functions namespaced by the keywords and drawn from the composers' functions.

Parameters **namespaces** – Composers that will be added at the namespace that corresponds to the arguments key

Returns A new Composer with all the input composers functions added as namespaces.

update_parameters(***parameters*) → fn_graph.Composer
Allows you to pass static parameters to the graph, they will be exposed as callables.

update_tests(***tests*) → fn_graph.Composer
Adds tests to the composer.

A test is a function that should check a property of the calculations results and raise an exception if they are not met.

The work exactly the same way as functions in terms of resolution of arguments. They are run with the run_test method.

update_without_prefix(*prefix: str*, **functions*, ***kwargs*) → fn_graph.Composer
Given a prefix and a list of (named) functions, this adds the functions to the composer but first strips the prefix from their name. This is very useful to stop name shadowing.

Parameters

- **prefix** – The prefix to strip off the function names
- **functions** – functions to add while stripping the prefix
- **kwargs** – named functions to add

Returns A new composer with the functions added

update_without_suffix(*suffix: str*, **functions*, ***kwargs*) → fn_graph.Composer
Given a suffix and a list of (named) functions, this adds the functions to the composer but first strips the suffix from their name. This is very useful to stop name shadowing.

Parameters

- **suffix** – The suffix to strip off the function names
- **functions** – functions to add while stripping the suffix
- **kwargs** – named functions to add

Returns A new composer with the functions added

fn_graph.**ComposerTestResult**
The results of Composer.run_tests()

alias of `fn_graph.TestResult`

Python Module Index

f

Index