nodes Documentation

Release alpha

Adam M. Donahue

June 21, 2015

Contents

An easy-to-use graph-oriented object model for Python.

CHAPTER 1

Overview

Graph-oriented programming, a functional reactive model in which changes to function inputs trigger the need for reevaluation of those functions, is generally a feature reserved for languages with strong functional programming support, particular those that perform strong type checking and that can enforce function purity at compile time.

This is too bad. A graph-oriented programming model, in addition to providing a useful model for the relationships between objects in your system, can be an extremely productivity-enhancing tool, freeing the developer from a lot of the coding that would typically be needed to perform such things as memoization, lazy evaluation, dependency tracking, subscriptions and callbacks, temporary changes, report building, and so forth.

It doesn't have to be this way.

A programmer can still leverage a graph-oriented model in a less strictly typed language as long as she adheres to the semantics required by such a model (mainly that on-graph functions must be pure and side-effect free).

The goal of nodes is to bridge this gap by providing Python developers with a simple, elegant way to put their classes on a graph.

CHAPTER 2

Features

- Ease of use.
- Dependency tracking and invalidation for on-graph nodes.
- Lazy evaluation.
- Memoization.
- Change delegation.
- Contextual evaluation. (What-if scenario building.)

CHAPTER 3

Current Limitations

- Runtime overhead. The current version focuses on the developer interface, and is not tuned for high performance. So there is overhead associated with each on-graph method that will impact programs that require high performance.
- Single threaded. The graph is single threaded, and use within a multiple threaded environment is not yet supported.
- No object persistence. The graph must be constructed in memory each time a program is launched; there is no object persistence layer yet. *This is a feature under development, namely, an object-oriented database of Python objects allowing one to save and read back on-graph objects.*
- Dynamic graph construction. There are two approaches I could have taken to building the graph. One involves using a AST to determine the full structure of the graph upfront. The other involves dynamically discovering the graph as graph methods are called.

  At present I use the dynamic route, which means that graph edges are added and updated as on-graph functions are called. I plan to abstract the way the graph is discovered into a separate class and allow a user to perform static graph discovery vs dynamic, if desired.

  (One benefit to static discovery is that it makes it possible to query the graph about its relationships without having to had evaluated its functions, and even in that case, in a dynamic graph that has been evaluated with some set of inputs, one still doesn't necessarily get a full picture of the graph, but instead sees the relationships as of a state in time - thatis, how the graph was used before the time at which you asked it for its structure.)

CHAPTER 4

Using nodes: Requirements

Putting an object on the graph is easy, as this example illustrates. There are three things a developer must do.

Two of these are technical:

- His class must be a subclass of GraphObject.
- On-graph methods in this class must be decorated with @graphMethod.

The third is has to do with the semantics of the methods he has decorated with @graphMethod. All graph methods must be *pure and side-effect free* (with some exceptions that we need not delve into here).

A function is *pure* if given the same inputs it returns the same output:

```
def cubed(self, x):
    return x * x * x
```

is a pure function, whereas:

```
file.readline
```

is not.

A function has *side-effects* if it modifies global state in any way. cubed is side-effect free, but file.readline is not, as it would update state indicating where in the file its next read should occur.

Purity and lack of side-effects often go hand-in-hand, and vice-versa.

Why is purity so important? Because without purity we cannot know when a function's value needs to be recomputed; if that value is determined by factors other than the inputs to the function then we lose control over when a function needs to be invalidated. And a huge benefit of the graph is its support for automatic node invalidation, node memoization, and lazy evaluation of node functions.

(The documentation will contain more detail on the model used and the patterns one can leverage to perform common operations in a graph-consistent manner. I don't want to get too Haskell-y on you because this is a Python module, not a Haskell library.)

So you may be thinking, jeez, this all sounds quite mathy and it sounds as if it'll be a pain to write proper on-graph methods.

That's a fair initial reaction. But the reality is that developing with nodes is not that diffcult, and if you follow its practices you end up with cleaner code and maybe even a new way of thinking about problems.

CHAPTER 5

Using nodes: An Example

The following example illustrates how you might use nodes to put a simple example class on the graph. It doesn't cover all of nodes' features but will give you an idea of its flavor.

The comments below indicate the status of each graph method after a given calculation. At this point I'm going to switch to using the term "node" instead of method, as in reality a method may map to multiple nodes (for example, in the case where the method has arguments in addition to self).

- invalid: The node is not set and the method body will run when its value is next requested.
- calced: The node is valid and its value was calculated by running the function body and memoizing the result. As long as the node remains valid its memoized output will be returned with no recomputation required.
- set: The node was set to a specific value by the user. This setting is non-contextual (global) to the graph.
- overlaid: The node was overlaid to a specific value by the user within a GraphContext. The overlay is active only within the context, and upon exiting the context the node's state is reverted to its prior value. (This is not strictly true; if global dependencies changed that were hidden by the context the node might have been invalidated outside the context and thus require computation the next time it's valid is requested.)

That said, here is the code:

```
class Example(nodes.GraphObject):

    @nodes.graphMethod
    def X(self):
        return 'X:%s:%s' % (self.Y(), self.Z())

    @nodes.graphMethod(nodes.Settable)
    def Y(self):
        return 'Y'

    @nodes.graphMethod(nodes.Settable)
    def Z(self):
        return 'Z'

def main():
    example = Example()

    # example.X             <invalid>
    # example.Y             <invalid>
    # example.Z             <invalid>

    example.X()
```

```
# example.X == 'X:Y:Z'  <calced>
# example.Y == 'Y'      <calced>
# example.Z == 'Z'      <calced>

example.Y = 'y'

# example.X             <invalid>
# example.Y == 'y'      <set>
# example.Z == 'Z'      <calced>

example.X()

# example.X == 'X:y:Z'  <calced>
# example.Y == 'y'      <set>
# example.Z == 'Z'      <calced>

example.Y.clearValue()

# example.X             <invalid>
# example.Y             <invalid> (maybe)
# example.Z == 'Z'      <calced>

example.X()

# example.X == 'X:Y:Z'  <calced>
# example.Y == 'Y'      <calced>
# example.Z == 'Z'      <calced>

with nodes.GraphContext():
    example.Y.overlayValue('y')

    # example.X             <invalid>
    # example.Y == 'Y'      <overlaid>
    # example.Z == 'z'      <calced>

    example.X()

    # example.X == 'X:Y:z'  <calced>
    # example.Y == 'Y'      <overlaid>
    # example.Z == 'z'      <calced>

# example.X             <invalid> (maybe)
# example.Y == 'Y'      <invalid> (maybe)
# example.Z == 'Z'      <calced>

with nodes.GraphContext() as savedContext:
    example.Y.overlayValue('y')

    # example.X         <invalid>
    # example.Y == 'y'  <overlaid>
    # example.Z == 'Z'  <calced>

# example.X             <invalid (maybe)>
# example.Y             <invalid (maybe)>
# example.Z == 'Z'      <calced>

example.X()
```

```
    # example.X == 'X:Y:Z'  <calced>
    # example.Y == 'Y'      <calced>
    # example.Z == 'Z'      <calced>

    with savedContext:

        # example.X         <invalid>
        # example.Y == 'y'  <overlaid>
        # example.Z == 'Z'  <calced>

        example.X()

        # example.X == 'X:y:Z'    <calced>
        # example.Y == 'y'        <overlaid>
        # example.Z == 'Z'        <calced>

        with nodes.GraphContext():
            example.Z.overlayValue('z')

            # example.X        <invalid>
            # example.Y == 'y' <overlaid>
            # example.Z == 'z' <overlaid>

            example.X()

            # example.X == 'X:y:z'    <calced>
            # example.Y == 'y'        <overlaid>
            # example.Z == 'z'        <overlaid>

        # example.X         <invalid>
        # example.Y == 'y'  <overlaid>
        # example.Z == 'Z'  <invalid (maybe)>
```