graphi Documentation

Release 0.2.0

Max Fischer

Jul 26, 2018

Documentation Topics Overview:

1	Quick Usage Reference	3
2	chainlet Changelog	5
3	graphi	7
4	Frequently Asked Questions	15
5	Indices and tables	17
	Python Module Index	19

CHAPTER 1

Quick Usage Reference

GraphI is primarily meant for working directly on graph data. The primitives you need to familiarise yourself with are

1. graphs, which are extended containers,
2. nodes, which are arbitrary objects in a graph,
3. edges, which are connections between objects in a graph, and
4. edge values, which are values assigned to connections in a graph.

$$\underbrace{\texttt{flighttime}}_{\texttt{graph}}[\overbrace{\underbrace{\texttt{Berlin}}_{\texttt{node}} : \underbrace{\texttt{London}}_{\texttt{node}}}^{\texttt{edge}}] = \underbrace{3900}_{\texttt{value}}$$

This documentation page gives an overview of the most important aspects. The complete interface of GraphI is defined and documented by *Graph*.

1.1 Creating Graphs and adding Nodes

You can create graphs empty, via cloning, from nodes or with nodes, edges and values. For many use-cases, it is simplest to start with a set of nodes:

```python
from graphi import graph

planets = graph("Earth", "Jupiter", "Mars", "Pluto")
```

Once you have a graph, it works similar to a set for nodes. You can simply *add()* and *discard()* nodes:

```python
planets.add("Venus")
planets.add("Mercury")
planets.discard("Pluto")
```

1.2 Working with Edges and Values

To really make use of a graph, you will want to add edges and give them values. Simply pick a connection *from* a node *to* a node and assign it a value:

```
# store the average distance between planets
planets["Earth":"Venus"] = 41400000
```

An edge is always of the form `start:end`, but values can be of arbitrary type. For example, you can easily add multiple values for a single edge using containers:

```
# add multiple values as an implicit tuple
planets["Earth":"Venus"] = 41400000, 258000000
# add multiple values as an explicit, mutable list
planets["Earth":"Mars"] = [78000000, 378000000]
```

The `:`-syntax of edges is not just pretty - it ensures that you never, ever accidentally mix up nodes and edges. This allows you to safely use the same `graph[item]` interface for nodes and edges.

If you need to define an edge outside of graph accesses, explicitly use *`Edge`*:

```
from graphi import Edge

if Edge["Venus":"Earth"] in planets:
    print("Wait, isn't there a pattern for this?")
```

1.3 Graphs as Python Containers

`GraphI` is all about letting you handle graphs with well-known interfaces. A graph is a container indexed by either nodes or edges:

```
print(planets["Venus":"Earth"])
del planets["Jupiter"]
```

Even though it contains nodes, edges and values, it presents its nodes first - similar to keys in a `dict`. However, you can efficiently access its various elements via views:

```
print("My father only told me about %d of our planets." % len(planets))
print("But I looked up %d distances between planets:" % len(planets.edges())
for planet_a, planet_b, distances in planets.items():
    print("  %s to %s: %s" % (planet_a, planet_b, '-'.join(distances)))
```

CHAPTER 2

chainlet Changelog

2.1 0.2.0 2017-07-31

Notes Definition of primary interface, algorithms (`Graph.neighbours`) will be revised

New Features Added AdjacencyGraph

Major Changes Defined graph container interface

Minor Changes Added documentation

CHAPTER 3

graphi

3.1 graphi package

3.1.1 Subpackages

graphi.compatibility package

`graphi.compatibility.`**`compat_version = sys.version_info(major=2, minor=7, micro=12, release`**

python version this module has been finalized for

Submodules

graphi.compatibility.python2 module

`class` `graphi.compatibility.python2.`**`ABCBase`**

Bases: `object`

Helper class that provides a standard way to create an ABC using inheritance.

A helper class that has `ABCMeta` as its metaclass. With this class, an abstract base class can be created by simply deriving from `ABC`, avoiding sometimes confusing metaclass usage.

Note that the type of `ABC` is still `ABCMeta`, therefore inheriting from `ABC` requires the usual precautions regarding metaclass usage, as multiple inheritance may lead to metaclass conflicts.

New in version 3.4.

Changed in version 3.3: Subclasses can use `register()` as a Decorator.

`classmethod register`(*subclass*)

Register *subclass* as a "virtual subclass" of this ABC.

Changed in version 3.3: Returns the registered subclass, to allow usage as a class decorator.

graphi.compatibility.python3 module

graphi.types package

Submodules

graphi.types.adjacency_graph module

3.1.2 Submodules

graphi.abc module

class graphi.abc.**AdjacencyList**

Bases: dict, _abcoll.MutableMapping

Edge values of nodes to a node in a graph

This represents edges in a `graph` originating from `node` as a mapping to their values. For example, the edge `graph[a:b] = c` corresponds to `adjacency[b] = c` for node `a`.

exception graphi.abc.**AdjacencyListTypeError**(*item*)

Bases: exceptions.TypeError

AdjacencyList set with an incorrect type

exception graphi.abc.**EdgeError**

Bases: exceptions.Exception

Graph edge not found

class graphi.abc.**EdgeView**(*graph*)

Bases: *graphi.abc.GraphView*

View on the edges in a graph

class graphi.abc.**Graph**(**source*, ***kwargs*)

Bases: _abcoll.Container

Abstract Base Class for graphs representing values of edges between nodes

A *Graph* is a container for primitive nodes and edges. There are three types of elements handled by a graph:

1. primitive *nodes*,
2. slice-like *edges* as pairs of nodes, and
3. primitive *edge values*.

Both nodes and edge values are conceptually similar to keys and values of `dict`. However, the concept of node pairs as edges adds additional functionality. The fixed relation between arbitrary nodes `a, b` and the directed pair `a:b` creates two value-type layers:

1. each *node* is mapped to all its outgoing edges,
2. each *edge* is mapped to the respective edge value.

In short, `graph[a]` provides a collection of edges originating at `a`, while `graph[a:b]` provides the specific edge value from `a` to `b`.

Note: Many interfaces return the rich *`Edge`* type for its added usability. To access an edge value, using `slice` such as `graph[a:b]` is sufficient, however.

Similar to `Mappings`, nodes are the primary keys of a *`Graph`*. As a result, the container interfaces, such as `iter` and `len`, operate on nodes. In general, nodes can be of arbitrary type as long as they are hashable.

By default, edges in a *`Graph`* are directed and unique: The edges represented by `graph[a:b]` and `graph[b:a]` are separate with opposite direction. Each edge is unique, i.e. there is only one edge `graph[a:b]`. A loop is represented by the edge `graph[a:a]`. The edge entities stored in the graph may be arbitrary objects.

As such, the interface of *`Graph`* defaults to describing a directed graph. However, other types of graph can be expressed as well. These generally do not form separate types in term of implementation.

Multigraphs allow for multiple edges between pairs of nodes. In this case, all edge values are containers (such as `list` or `set`) of arbitrary size. Whether a *`Graph`* is a graph of containers or a multigraph depends on the context.

Undirected Graphs do not distinguish between `graph[a:b]` and `graph[b:a]`. This can be enforced by symmetry of edge values, which guarantees that `graph[a:b] == graph[b:a]` always applies.

`g.undirected`

Indicates whether *`Graph`* `g` is guaranteed to be undirected, having only symmetric edge values. If `True`, `g[a:b] is g[b:a]` for any nodes `a` and `b` in `g`; the graph enforces this, e.g. `g[a:b] = c` implies `g[b:a] = c`. If `False`, symmetric edges are allowed but not enforced.

Read-only unless explicitly indicated otherwise.

There are several ways to initialise a new graph; their main difference is which element types are left empty.

`Graph()`

Create a new empty graph. No nodes, edges or values are filled in.

`Graph(graph)`

Create a new graph with all nodes, edges and values of `graph`. The resulting graph is a shallow copy of `graph` - the identity of elements is preserved.

`Graph(a, b, c, ...)`
`Graph([a, b, c, ...])`
`Graph({a, b, c, ...})`
`Graph(<iterable for a, b, c, ...>)`

Create a new graph with nodes `a`, `b`, `c`, `d`, and so on. No edges or values are created explicitly.

`Graph({a: {b: ab_edge, c: ...}, b: {a: ab_edge, ...}})`
`Graph({a: AdjacencyList({b: ab_edge, c: ...}), b: AdjacencyList(...), ...})`

Create a new graph with nodes `a`, `b`, `c`, and so on. Initialize edges to `graph[a:b] = ab_edge`, `graph[b:a] = ba_edge`, and so on.

Note: If only a single argument is provided, graph and mapping initialization is preferred over iterable initialisation. To initialize a graph with a graph or mapping as the sole node, wrap it in an iterable, e.g. `Graph([graph])`.

All implementations of this ABC guarantee the following operators:

`len(g)`

Return the number of nodes in the graph `g`.

`g[a:b]`
Return the value of the edge between nodes `a` and `b`. Raises *`EdgeError`* if no edge is defined for the nodes. Undirected graphs guarantee `g[a:b] == g[b:a]`.

`g[a:b] = value`
Set the value of the edge between nodes `a` and `b` to `value` for graph `g`.

`del g[a:b]`
Remove the edge and value between nodes `a` and `b` from `g`. Raises *`EdgeError`* if the edge is not in the graph.

`g[a]`
Return the edges between nodes `a` and any other node as an *`AdjacencyList`* corresponding to `{b: ab_edge, c: ac_edge, ...}`. Raises *`NodeError`* if `a` is not in `g`.

`g[a] = None`
`g[a] = a`
`g.add(a)`
Add the node `a` to graph `g` if it does not exist. Do not add, remove or modify existing edges. Graphs for which edges are computed, not set, may create them implicitly.

`g[a] = {}`
`g[a] = AdjacencyList()`
Add the node `a` to graph `g` if it does not exist. Remove any existing edges originating at `a` from graph `g`.

`g[a] = {b: ab_edge, c: ac_edge, ...}`
`g[a] = AdjacencyList(b=ab_edge, c=c_edge)`
Add the node `a` to graph `g` if it does not exist. Set the value of the edge between nodes `a` and `b` to `ab_edge`, between `a` and `c` to `ac_edge`, and so on. Remove any other edge from `a`. Raises *`NodeError`* if any of `b`, `c`, etc. are not in `g`.

`del g[a]`
Remove the node `a` and all its edges from `g`. Raises *`NodeError`* if the node is not in the graph.

`a in g`
Return `True` if `g` has a node `a`, else `False`.

`Edge[a:b] in g`
`Edge(a, b) in g`
Return `True` if `g` has an edge from node `a` to `b`, else `False`.

`iter(g)`
Return an iterator over the nodes in `g`.

In addition, several methods are provided. While methods and operators for retrieving data must be implemented by all subclasses, methods for modifying data may not be applicable to certain graphs.

`add`(*node*)
Safely add a node to the graph, without modifying existing edges

If the `node` is not part of the graph, it is added without any explicit edges. If the node is already present, this has no effect.

Note: Graphs which compute edges may implicitly create new edges if `node` is new to the graph.

`clear`()
Remove all elements from this graph

`copy`()
Return a shallow copy of this graph

discard(*item*)
Remove a node or edge from the graph if it is a member

Parameters **item** – node or edge to discard from the graph

edges()
Return a new view of the graph's edges

Returns view of the graph's edges

Return type *EdgeView*

get(*item*, *default=None*)
Return the value for node or edge `item` if it is in the graph, else default. If `default` is not given, it defaults to `None`, so that this method never raises a *NodeError* or *EdgeError*.

Parameters

- **item** – node or edge to look up in the graph
- **default** – default to return if `item` is not in the graph

items()
Return a new view of the graph's edges and their values

Returns view of the graph's edges and their values

Return type *ItemView*

neighbourhood(*node*, *distance=None*)
Yield all nodes to which there is an edge from `node` in the graph

Parameters

- **node** – node from which edges originate.
- **distance** – optional maximum distance to other nodes.

Returns iterator of neighbour nodes

Raises ***NodeError*** – if `node` is not in the graph

When `distance` is not `None`, it is the maximum allowed edge value. This is interpreted using the `<=` operator as `graph[edge] <= distance`.

If there is a valid edge `graph[node:node] <= distance`, then `node` is part of its own neighbourhood.

nodes()
Return a new view of the graph's nodes

Returns view of the graph's nodes

Return type *NodeView*

undirected = False
whether this graph is undirected, having only symmetric edges

update(*other*)
Update the graph with the nodes, edges and values from `other`, overwriting existing elements.

Parameters **other** (*Graph* or *ItemView*) – graph or items from which to pull elements

values()
Return a new view of the values of the graph's edges

Returns view of the values of the graph's edges

Return type `ValueView`

class `graphi.abc.`**`GraphView`**`(`*graph*`)`

Bases: `_abcoll.Sized`

Dynamic view on the content of a `Graph`

View objects represent a portion of the content of a graph. A view allows to work with its scope without copying the viewed content. It is dynamic, meaning that any changes to the graph are reflected by the view.

Each view works only on its respective portion of the graph. For example, `edge in nodeview` will always return `False`.

`len(graphview)`

Return the number of nodes, node pairs or edges in the graph.

`x in graphview`

Return `True` if x is a node, node pair or edge of the graph.

`iter(graphview)`

Return an iterator over the nodes, node pairs or edges in the graph.

Each view strictly defines the use of nodes, edges or values. As such, edges are safely represented as a tuple of start and end node.

`undirected`

class `graphi.abc.`**`ItemView`**`(`*graph*`)`

Bases: `graphi.abc.GraphView`

View on the edges and values in a graph

Represents edges and their value as a `tuple` of `(tail, head, value)`. For example, the edge `graph[a:b] = c` corresponds to the item `(a, b, c)`.

exception `graphi.abc.`**`NodeError`**

Bases: `exceptions.Exception`

Graph node not found

class `graphi.abc.`**`NodeView`**`(`*graph*`)`

Bases: `graphi.abc.GraphView`

View on the nodes of a graph

class `graphi.abc.`**`ValueView`**`(`*graph*`)`

Bases: `graphi.abc.GraphView`

View on the values of edges in a graph

graphi.edge module

class `graphi.edge.`**`Edge`**`(`*start*, *stop*, *step=None*`)`

Bases: `object`

An edge in a graph as a pair of nodes

Parameters

- **`start`** – the start or tail of an edge
- **`stop`** – the stop or head of an edge
- **`step`** – currently unused

This is a verbose interface for creating edges between nodes for use in a graph. It allows using slice notation independent of a graph:

```
>>> atb = Edge[a:b]
>>> a2b = Edge(a, b)
>>> graph[a2b] = 1337
>>> graph[a:b] == graph[atb] == graph[a2b] == graph[Edge[a:b]] == graph[Edge(a,␣
↪b)]
True
```

A *Edge* can also be used for explicit containment tests:

```
>>> Edge[a:b] in graph
True
```

In addition to their slice-like nature, *Edge* is iterable and indexable. This allows for easy unpacking:

```
>>> edge = Edge[a:b]
>>> tail, head = edge
```

Note: This class creates a representation of an edge as a connection between nodes. Edge *values* can be arbitrary objects.

Warning: Even though *Edge* behaves like a `slice` in graphs, builtin containers such as `list` cannot use a *Edge*.

start

stop

class graphi.edge.**EdgeMeta**

Bases: `type`

Metaclass for *Edge* to support `Edge[a:b]`

`GraphI` is a lightweight graph library - it is suitable to model networks, connections and other relationships. Compared to other graph libraries, `GraphI` aims for being as pythonic as possible. If you are comfortable using `list`, `dict` or other types, `GraphI` is intuitive and straight-forward to use.

```
# create a graph with initial nodes
airports = Graph("New York", "Rio", "Tokyo")
# add connections between nodes
airports["New York":"Rio"] = timedelta(hours=9, minutes=50)
airports["New York":"Tokyo"] = timedelta(hours=13, minutes=55)
```

At its heart, `GraphI` is built to integrate with Python's data model. It natively works with primitives, iterables, mappings and whatever you need. For example, creating a multigraph is as simple as using multiple edge values:

```
# add multiple connections between nodes
airports["Rio":"Tokyo"] = timedelta(days=1, hours=2), timedelta(days=1, hours=3)
```

With its general-purpose design, `GraphI` makes no assumptions about your data. You are free to use whatever is needed to solve your problem, not please data structure.

CHAPTER 4

Frequently Asked Questions

Yet another graph library? The goal of `GraphI` is not to be another graph library, but to provide an intuitive graph interface. Working with complex graphs should be as easy *for you* as working with any other primitive type.

Where are all the algorithms? First and foremost, `GraphI` is designed for you to *work on graph data* instead of pre-sliced storybook data. `GraphI` implements only algorithms that

1. are fundamental building blocks for advanced algorithms, and/or
2. benefit from knowledge of internal data structures.

What about performance? At its core, `GraphI` uses Python's native, highly optimized data structures. For any non-trivial graph algorithm, the provided performance is more than sufficient.

From our experience, performance critical code is best run with PyPy. This will not just optimize isolated pieces, but the actual combination of your algorithm and `GraphI` as a whole.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Documentation built from graphi 0.2.0 at Jul 26, 2018.

Python Module Index

g

graphi, 7
graphi.abc, 8
graphi.compatibility, 7
graphi.compatibility.python2, 7
graphi.edge, 12
graphi.types, 8

Index

A

ABCBase (class in graphi.compatibility.python2), 7
add() (graphi.abc.Graph method), 10
AdjacencyList (class in graphi.abc), 8
AdjacencyListTypeError, 8

C

clear() (graphi.abc.Graph method), 10
compat_version (in module graphi.compatibility), 7
copy() (graphi.abc.Graph method), 10

D

discard() (graphi.abc.Graph method), 10

E

Edge (class in graphi.edge), 12
EdgeError, 8
EdgeMeta (class in graphi.edge), 13
edges() (graphi.abc.Graph method), 11
EdgeView (class in graphi.abc), 8

G

get() (graphi.abc.Graph method), 11
Graph (class in graphi.abc), 8
graphi (module), 7
graphi.abc (module), 8
graphi.compatibility (module), 7
graphi.compatibility.python2 (module), 7
graphi.edge (module), 12
graphi.types (module), 8
GraphView (class in graphi.abc), 12

I

items() (graphi.abc.Graph method), 11
ItemView (class in graphi.abc), 12

N

neighbourhood() (graphi.abc.Graph method), 11
NodeError, 12
nodes() (graphi.abc.Graph method), 11
NodeView (class in graphi.abc), 12

R

register() (graphi.compatibility.python2.ABCBase class method), 7

S

start (graphi.edge.Edge attribute), 13
stop (graphi.edge.Edge attribute), 13

U

undirected (graphi.abc.Graph attribute), 11
undirected (graphi.abc.GraphView attribute), 12
update() (graphi.abc.Graph method), 11

V

values() (graphi.abc.Graph method), 11
ValueView (class in graphi.abc), 12