altgraph Documentation

Release 0.17.3

Ronald Oussoren

Sep 25, 2022

Contents

altgraph is a fork of graphlib: a graph (network) package for constructing graphs, BFS and DFS traversals, topological sort, shortest paths, etc. with graphviz output.

The primary users of this package are macholib and modulegraph.

CHAPTER 1

Release history

1.1 0.17.3

- Update classifiers for Python 3.11

1.2 0.17.2

- Change in setup.py to fix the sidebar links on PyPI

1.3 0.17.1

- Explicitly mark Python 3.10 as supported in wheel metadata.

1.4 0.17

- Explicitly mark Python 3.8 as supported in wheel metadata.
- Migrate from Bitbucket to GitHub
- Run black on the entire repository

1.5 0.16.1

- Explicitly mark Python 3.7 as supported in wheel metadata.

1.6 0.16

- Add LICENSE file

1.7 0.15

- `ObjectGraph.get_edges`, `ObjectGraph.getEdgeData` and `ObjectGraph.updateEdgeData` accept *None* as the node to get and treat this as an alias for *self* (as other methods already did).

1.8 0.14

- Issue #7: Remove use of `iteritems` in altgraph.GraphAlgo code

1.9 0.13

- Issue #4: Graph._bfs_subgraph and back_bfs_subgraph return subgraphs with reversed edges

  Fix by "pombredanne" on bitbucket.

1.10 0.12

- Added `ObjectGraph.edgeData` to retrieve the edge data from a specific edge.
- Added `AltGraph.update_edge_data` and `ObjectGraph.updateEdgeData` to update the data associated with a graph edge.

1.11 0.11

- Stabilize the order of elements in dot file exports, patch from bitbucket user 'pombredanne'.
- Tweak setup.py file to remove dependency on distribute (but keep the dependency on setuptools)

1.12 0.10.2

- There where no classifiers in the package metadata due to a bug in setup.py

1.13 0.10.1

This is a bugfix release

Bug fixes:

- Issue #3: The source archive contains a README.txt while the setup file refers to ReadMe.txt.

  This is caused by a misfeature in distutils, as a workaround I've renamed ReadMe.txt to README.txt in the source tree and setup file.

1.14 0.10

This is a minor feature release

Features:

- Do not use "2to3" to support Python 3.

  As a side effect of this altgraph now supports Python 2.6 and later, and no longer supports earlier releases of Python.

- The order of attributes in the Dot output is now always alphabetical.

  With this change the output will be consistent between runs and Python versions.

1.15 0.9

This is a minor bugfix release

Features:

- Added `altgraph.ObjectGraph.ObjectGraph.nodes`, a method yielding all nodes in an object graph.

Bugfixes:

- The 0.8 release didn't work with py2app when using python 3.x.

1.16 0.8

This is a minor feature release. The major new feature is a extensive set of unittests, which explains almost all other changes in this release.

Bugfixes:

- Installing failed with Python 2.5 due to using a distutils class that isn't available in that version of Python (issue #1 on the issue tracker)
- `altgraph.GraphStat.degree_dist` now actually works
- `altgraph.Graph.add_edge(a, b, create_nodes=False)` will no longer create the edge when one of the nodes doesn't exist.
- `altgraph.Graph.forw_topo_sort` failed for some sparse graphs.
- `altgraph.Graph.back_topo_sort` was completely broken in previous releases.
- `altgraph.Graph.forw_bfs_subgraph` now actually works.
- `altgraph.Graph.back_bfs_subgraph` now actually works.
- `altgraph.Graph.iterdfs` now returns the correct result when the `forward` argument is `False`.
- `altgraph.Graph.iterdata` now returns the correct result when the `forward` argument is `False`.

Features:

- The `altgraph.Graph` constructor now accepts an argument that contains 2- and 3-tuples instead of requiring that all items have the same size. The (optional) argument can now also be any iterator.
- `altgraph.Graph.Graph.add_node` has no effect when you add a hidden node.
- The private method `altgraph.Graph._bfs` is no longer present.
- The private method `altgraph.Graph._dfs` is no longer present.
- `altgraph.ObjectGraph` now has a `__contains__` methods, which means you can use the `in` operator to check if a node is part of a graph.
- `altgraph.GraphUtil.generate_random_graph` will raise `GraphError` instead of looping forever when it is impossible to create the requested graph.
- `altgraph.Dot.edge_style` raises `GraphError` when one of the nodes is not present in the graph. The method silently added the tail in the past, but without ensuring a consistent graph state.
- `altgraph.Dot.save_img` now works when the mode is `"neato"`.

1.17 0.7.2

This is a minor bugfix release

Bugfixes:

- distutils didn't include the documentation subtree

1.18 0.7.1

This is a minor feature release

Features:

- Documentation is now generated using sphinx and can be viewed at <http://packages.python.org/altgraph>.
- The repository has moved to bitbucket
- `altgraph.GraphStat.avg_hops` is no longer present, the function had no implementation and no specified behaviour.
- the module `altgraph.compat` is gone, which means altgraph will no longer work with Python 2.3.

1.19 0.7.0

This is a minor feature release.

Features:

- Support for Python 3
- It is now possible to run tests using 'python setup.py test'

  (The actual testsuite is still very minimal though)

CHAPTER 2

License

2.1 MIT License

CHAPTER 3

altgraph — A Python Graph Library

altgraph is a fork of graphlib tailored to use newer Python 2.3+ features, including additional support used by the py2app suite (modulegraph and macholib, specifically).

altgraph is a python based graph (network) representation and manipulation package. It has started out as an extension to the graph_lib module written by Nathan Denny it has been significantly optimized and expanded.

The *altgraph.Graph.Graph* class is loosely modeled after the LEDA (Library of Efficient Datatypes) representation. The library includes methods for constructing graphs, BFS and DFS traversals, topological sort, finding connected components, shortest paths as well as a number graph statistics functions. The library can also visualize graphs via graphviz.

exception altgraph.**GraphError**
Exception raised when methods are called with bad values of an inconsistent state.

CHAPTER 4

altgraph.Graph — Basic directional graphs

The module *altgraph.Graph* provides a class *Graph* that represents a directed graph with *N* nodes and *E* edges.

class altgraph.Graph.**Graph**([*edges*])
Constructs a new empty *Graph* object. If the optional *edges* parameter is supplied, updates the graph by adding the specified edges.

All of the elements in *edges* should be tuples with two or three elements. The first two elements of the tuple are the source and destination node of the edge, the optional third element is the edge data. The source and destination nodes are added to the graph when the aren't already present.

4.1 Node related methods

Graph.**add_node**(*node*[, *node_data*])
Adds a new node to the graph if it is not already present. The new node must be a hashable object.

Arbitrary data can be attached to the node via the optional *node_data* argument.

Note: the node also won't be added to the graph when it is present but currently hidden.

Graph.**hide_node**(*node*)
Hides a *node* from the graph. The incoming and outgoing edges of the node will also be hidden.

Raises *altgraph.GraphError* when the node is not (visible) node of the graph.

Graph.**restore_node**(*node*)
Restores a previously hidden *node*. The incoming and outgoing edges of the node are also restored.

Raises *altgraph.GraphError* when the node is not a hidden node of the graph.

Graph.**restore_all_nodes**()
Restores all hidden nodes.

Graph.**number_of_nodes**()
Return the number of visible nodes in the graph.

Graph.**number_of_hidden_nodes**()
Return the number of hidden nodes in the graph.

Graph.**node_list**()
Return a list with all visible nodes in the graph.

Graph.**hidden_node_list**()
Return a list with all hidden nodes in the graph.

altgraph.Graph.**node_data**(*node*)
Return the data associated with the *node* when it was added.

Graph.**describe_node**(*node*)
Returns *node*, the node's data and the lists of outgoing and incoming edges for the node.

Note: the edge lists should not be modified, doing so can result in unpredicatable behavior.

Graph.**__contains__**(*node*)
Returns True iff *node* is a node in the graph. This method is accessed through the *in* operator.

Graph.**__iter__**()
Yield all nodes in the graph.

Graph.**out_edges**(*node*)
Return the list of outgoing edges for *node*

Graph.**inc_edges**(*node*)
Return the list of incoming edges for *node*

Graph.**all_edges**(*node*)
Return the list of incoming and outgoing edges for *node*

Graph.**out_degree**(*node*)
Return the number of outgoing edges for *node*.

Graph.**inc_degree**(*node*)
Return the number of incoming edges for *node*.

Graph.**all_degree**(*node*)
Return the number of edges (incoming or outgoing) for *node*.

4.2 Edge related methods

Graph.**add_edge**(*head_id*, *tail_id*[, *edge data*[, *create_nodes*]])
Adds a directed edge from *head_id* to *tail_id*. Arbitrary data can be added via *edge_data*. When *create_nodes* is *True* (the default), *head_id* and *tail_id* will be added to the graph when the aren't already present.

Graph.**hide_edge**(*edge*)
Hides an edge from the graph. The edge may be unhidden at some later time.

Graph.**restore_edge**(*edge*)
Restores a previously hidden *edge*.

Graph.**restore_all_edges**()
Restore all edges that were hidden before, except for edges referring to hidden nodes.

Graph.**edge_by_node**(*head*, *tail*)
Return the edge ID for an edge from *head* to *tail*, or None when no such edge exists.

Graph.**edge_by_id**(*edge*)
Return the head and tail of the *edge*

Graph.**edge_data**(*edge*)
Return the data associated with the *edge*.

Graph.**update_edge_data**(*edge*, *data*)
Replace the edge data for *edge* by *data*. Raises KeyError when the edge does not exist.

New in version 0.12.

Graph.**head**(*edge*)
Return the head of an *edge*

Graph.**tail**(*edge*)
Return the tail of an *edge*

Graph.**describe_edge**(*edge*)
Return the *edge*, the associated data, its head and tail.

Graph.**number_of_edges**()
Return the number of visible edges.

Graph.**number_of_hidden_edges**()
Return the number of hidden edges.

Graph.**edge_list**()
Returns a list with all visible edges in the graph.

Graph.**hidden_edge_list**()
Returns a list with all hidden edges in the graph.

4.3 Graph traversal

Graph.**out_nbrs**(*node*)
Return a list of all nodes connected by outgoing edges.

Graph.**inc_nbrs**(*node*)
Return a list of all nodes connected by incoming edges.

Graph.**all_nbrs**(*node*)
Returns a list of nodes connected by an incoming or outgoing edge.

Graph.**forw_topo_sort**()
Return a list of nodes where the successors (based on outgoing edges) of any given node apear in the sequence after that node.

Graph.**back_topo_sort**()
Return a list of nodes where the successors (based on incoming edges) of any given node apear in the sequence after that node.

Graph.**forw_bfs_subgraph**(*start_id*)
Return a subgraph consisting of the breadth first reachable nodes from *start_id* based on their outgoing edges.

Graph.**back_bfs_subgraph**(*start_id*)
Return a subgraph consisting of the breadth first reachable nodes from *start_id* based on their incoming edges.

Graph.**iterdfs**(*start*[, *end*[, *forward*]])
Yield nodes in a depth first traversal starting at the *start* node.

If *end* is specified traversal stops when reaching that node.

If forward is True (the default) edges are traversed in forward direction, otherwise they are traversed in reverse direction.

Graph.**iterdata**(*start*[, *end*[, *forward*[, *condition*]]])
Yield the associated data for nodes in a depth first traversal starting at the *start* node. This method will not yield values for nodes without associated data.

If *end* is specified traversal stops when reaching that node.

If *condition* is specified and the condition callable returns False for the associated data this method will not yield the associated data and will not follow the edges for the node.

If forward is True (the default) edges are traversed in forward direction, otherwise they are traversed in reverse direction.

Graph.**forw_bfs**(*start*[, *end*])
Returns a list of nodes starting at *start* in some bread first search order (following outgoing edges).

When *end* is specified iteration stops at that node.

Graph.**back_bfs**(*start*[, *end*])
Returns a list of nodes starting at *start* in some bread first search order (following incoming edges).

When *end* is specified iteration stops at that node.

Graph.**get_hops**(*start*[, *end*[, *forward*]])
Computes the hop distance to all nodes centered around a specified node.

First order neighbours are at hop 1, their neigbours are at hop 2 etc. Uses *forw_bfs()* or *back_bfs()* depending on the value of the forward parameter.

If the distance between all neighbouring nodes is 1 the hop number corresponds to the shortest distance between the nodes.

Typical usage:

```
>>> print graph.get_hops(1, 8)
>>> [(1, 0), (2, 1), (3, 1), (4, 2), (5, 3), (7, 4), (8, 5)]
# node 1 is at 0 hops
# node 2 is at 1 hop
# ...
# node 8 is at 5 hops
```

4.4 Graph statistics

Graph.**connected**()
Returns True iff every node in the graph can be reached from every other node.

Graph.**clust_coef**(*node*)
Returns the local clustering coefficient of node.

The local cluster coefficient is the proportion of the actual number of edges between neighbours of node and the maximum number of edges between those nodes.

CHAPTER 5

altgraph.ObjectGraph — Graphs of objecs with an identifier

class altgraph.ObjectGraph.**ObjectGraph**([*graph*[, *debug*]])
A graph of objects that have a "graphident" attribute. The value of this attribute is the key for the object in the graph.

The optional *graph* is a previously constructed *Graph*.

The optional *debug* level controls the amount of debug output (see *msg()*, *msgin()* and *msgout()*).

Note: the altgraph library does not generate output, the debug attribute and message methods are present for use by subclasses.

ObjectGraph.**graph**
An *Graph* object that contains the graph data.

ObjectGraph.**addNode**(*node*)
Adds a *node* to the graph.

Note: re-adding a node that was previously removed using *removeNode()* will reinstate the previously removed node.

ObjectGraph.**createNode**(*self*, *cls*, *name*, **args*, ***kwds*)
Creates a new node using `cls(*args, **kwds)` and adds that node using *addNode()*.

Returns the newly created node.

ObjectGraph.**removeNode**(*node*)
Removes a *node* from the graph when it exists. The *node* argument is either a node object, or the graphident of a node.

ObjectGraph.**createReferences**(*fromnode*, *tonode*[, *edge_data*])
Creates a reference from *fromnode* to *tonode*. The optional *edge_data* is associated with the edge.

Fromnode and *tonode* can either be node objects or the graphident values for nodes. When *fromnode* is `None` *tonode* is a root for the graph.

altgraph.ObjectGraph.**removeReference**(*fromnode*, *tonode*)
Removes the reference from *fromnode* to *tonode* if it exists.

ObjectGraph.**getRawIdent**(*node*)
Returns the *graphident* attribute of *node*, or the graph itself when *node* is None.

altgraph.ObjectGraph.**getIdent**(*node*)
Same as getRawIdent(), but only if the node is part of the graph.

Node can either be an actual node object or the graphident of a node.

ObjectGraph.**findNode**(*node*)
Returns a given node in the graph, or Node when it cannot be found.

Node is either an object with a *graphident* attribute or the *graphident* attribute itself.

ObjectGraph.**__contains__**(*node*)
Returns True if *node* is a member of the graph. *Node* is either an object with a *graphident* attribute or the *graphident* attribute itself.

ObjectGraph.**flatten**([*condition*[, *start*]])
Yield all nodes that are entirely reachable by *condition* starting fromt he given *start* node or the graph root.

Note: objects are only reachable from the graph root when there is a reference from the root to the node (either directly or through another node)

ObjectGraph.**nodes**()
Yield all nodes in the graph.

ObjectGraph.**get_edges**(*node*)
Returns two iterators that yield the nodes reaching by outgoing and incoming edges for *node*. Note that the iterator for incoming edgets can yield None when the *node* is a root of the graph.

Use None for *node* to fetch the roots of the graph.

ObjectGraph.**filterStack**(*filters*)
Filter the ObjectGraph in-place by removing all edges to nodes that do not match every filter in the given filter list

Returns a tuple containing the number of: (*nodes_visited*, *nodes_removed*, *nodes_orphaned*)

ObjectGraph.edgeData(fromNode, toNode):
Return the edge data associated with the edge from *fromNode*
to *toNode*. Raises :exc:`KeyError` when no such edge exists.

ObjectGraph.**updateEdgeData**(*fromNode*, *toNode*, *edgeData*)
Replace the data associated with the edge from *fromNode* to *toNode* by *edgeData*.

Raises KeyError when the edge does not exist.

5.1 Debug output

ObjectGraph.**debug**
The current debug level.

ObjectGraph.**msg**(*level*, *text*, **args*)
Print a debug message at the current indentation level when the current debug level is *level* or less.

ObjectGraph.**msgin**(*level*, *text*, **args*)
Print a debug message when the current debug level is *level* or less, and increase the indentation level.

ObjectGraph.**msgout**(*level*, *text*, **args*)
Decrease the indentation level and print a debug message when the current debug level is *level* or less.

CHAPTER 6

altgraph.GraphAlgo — Graph algorithms

altgraph.GraphAlgo.**dijkstra**(*graph*, *start*[, *end*])
Dijkstra's algorithm for shortest paths.

Find shortest paths from the start node to all nodes nearer than or equal to the *end* node. The edge data is assumed to be the edge length.

Note: Dijkstra's algorithm is only guaranteed to work correctly when all edge lengths are positive. This code does not verify this property for all edges (only the edges examined until the end vertex is reached), but will correctly compute shortest paths even for some graphs with negative edges, and will raise an exception if it discovers that a negative edge has caused it to make a mistake.

---

altgraph.GraphAlgo.**shortest_path**(*graph*, *start*, *end*)
Find a single shortest path from the given start node to the given end node. The input has the same conventions as *dijkstra()*. The output is a list of the nodes in order along the shortest path.

CHAPTER 7

altgraph.GraphStat — Functions providing various graph statistics

The module *altgraph.GraphStat* provides function that calculate graph statistics. Currently there is only one such function, more may be added later.

altgraph.GraphStat.**degree_dist**(*graph*[, *limits*[, *bin_num*[, *mode*]]])

Groups the number of edges per node into *bin_num* bins and returns the list of those bins. Every item in the result is a tuple with the center of the bin and the number of items in that bin.

When the *limits* argument is present it must be a tuple with the mininum and maximum number of edges that get binned (that is, when *limits* is `(4, 10)` only nodes with between 4 and 10 edges get counted.

The *mode* argument is used to count incoming (`'inc'`) or outgoing (`'out'`) edges. The default is to count the outgoing edges.

CHAPTER 8

altgraph.GraphUtil — Utility functions

The module *altgraph.GraphUtil* performs a number of more or less useful utility functions.

altgraph.GraphUtil.**generate_random_graph**(*node_num, edge_num[, self_loops[, multi_edges]]*)

Generates and returns a *Graph* instance with *node_num* nodes randomly connected by *edge_num* edges.

When *self_loops* is present and True there can be edges that point from a node to itself.

When *multi_edge* is present and True there can be duplicate edges.

This method raises GraphError <altgraph.GraphError when a graph with the requested configuration cannot be created.

altgraph.GraphUtil.**generate_scale_free_graph**(*steps, growth_num*[, *self_loops*[, *multi_edges*]])

Generates and returns a *Graph* instance that will have *steps*growth_n um* nodes and a scale free (powerlaw) connectivity.

Starting with a fully connected graph with *growth_num* nodes at every step *growth_num* nodes are added to the graph and are connected to existing nodes with a probability proportional to the degree of these existing nodes.

> **Warning:** The current implementation is basically untested, although code inspection seems to indicate an implementation that is consistent with the description at Wolfram MathWorld

altgraph.GraphUtil.**filter_stack**(*graph*, *head*, *filters*)

Perform a depth-first oder walk of the graph starting at *head* and apply all filter functions in *filters* on the node data of the nodes found.

Returns (*visited*, *removes*, *orphans*), where

- *visited*: the set of visited nodes
- *removes*: the list of nodes where the node data doesn't match all *filters*.

- *orphans*: list of tuples (*last_good*, *node*), where node is not in *removes* and one of the nodes that is connected by an incoming edge is in *removes*. *Last_good* is the closest upstream node that is not in *removes*.

CHAPTER 9

altgraph.Dot — Interface to the dot language

The *Dot* module provides a simple interface to the file format used in the graphviz program. The module is intended to offload the most tedious part of the process (the **dot** file generation) while transparently exposing most of its features.

To display the graphs or to generate image files the graphviz package needs to be installed on the system, moreover the **dot** and **dotty** programs must be accesible in the program path so that they can be ran from processes spawned within the module.

9.1 Example usage

Here is a typical usage:

```
from altgraph import Graph, Dot

# create a graph
edges = [ (1,2), (1,3), (3,4), (3,5), (4,5), (5,4) ]
graph = Graph.Graph(edges)

# create a dot representation of the graph
dot = Dot.Dot(graph)

# display the graph
dot.display()

# save the dot representation into the mydot.dot file
dot.save_dot(file_name='mydot.dot')

# save dot file as gif image into the graph.gif file
dot.save_img(file_name='graph', file_type='gif')
```

9.2 Directed graph and non-directed graph

Dot class can use for both directed graph and non-directed graph by passing *graphtype* parameter.

Example:

```
# create directed graph(default)
dot = Dot.Dot(graph, graphtype="digraph")

# create non-directed graph
dot = Dot.Dot(graph, graphtype="graph")
```

9.3 Customizing the output

The graph drawing process may be customized by passing valid **dot** parameters for the nodes and edges. For a list of all parameters see the graphviz documentation.

Example:

```
# customizing the way the overall graph is drawn
dot.style(size='10,10', rankdir='RL', page='5, 5' , ranksep=0.75)

# customizing node drawing
dot.node_style(1, label='BASE_NODE',shape='box', color='blue' )
dot.node_style(2, style='filled', fillcolor='red')

# customizing edge drawing
dot.edge_style(1, 2, style='dotted')
dot.edge_style(3, 5, arrowhead='dot', label='binds', labelangle='90')
dot.edge_style(4, 5, arrowsize=2, style='bold')


.. note::

   dotty (invoked via :py:func:`~altgraph.Dot.display`) may not be able to
   display all graphics styles. To verify the output save it to an image
   file and look at it that way.
```

9.4 Valid attributes

- dot styles, passed via the *Dot.style()* method:

```
rankdir = 'LR'   (draws the graph horizontally, left to right)
ranksep = number (rank separation in inches)
```

- node attributes, passed via the *Dot.node_style()* method:

```
style = 'filled' | 'invisible' | 'diagonals' | 'rounded'
shape = 'box' | 'ellipse' | 'circle' | 'point' | 'triangle'
```

- edge attributes, passed via the `Dot.edge_style()` method:

```
style     = 'dashed' | 'dotted' | 'solid' | 'invis' | 'bold'
arrowhead = 'box' | 'crow' | 'diamond' | 'dot' | 'inv' | 'none' | 'tee' | 'vee'
weight    = number (the larger the number the closer the nodes will be)
```

- valid graphviz colors
- for more details on how to control the graph drawing process see the graphviz reference.

9.5 Class interface

class altgraph.Dot.**Dot**(*graph*[, *nodes*[, *edgefn*[, *nodevisitor*[, *edgevisitor*[, *name*[, *dot*[, *dotty*[, *neato*[, *graphtype*]]]]]]]]])

Creates a new Dot generator based on the specified *Graph*. The Dot generator won't reference the *graph* once it is constructed.

If the *nodes* argument is present it is the list of nodes to include in the graph, otherwise all nodes in *graph* are included.

If the *edgefn* argument is present it is a function that yields the nodes connected to another node, this defaults to `graph.out_nbr`. The constructor won't add edges to the dot file unless both the head and tail of the edge are in *nodes*.

If the *name* is present it specifies the name of the graph in the resulting dot file. The default is `"G"`.

The functions *nodevisitor* and *edgevisitor* return the default style for a given edge or node (both default to functions that return an empty style).

The arguments *dot*, *dotty* and *neato* are used to pass the path to the corresponding graphviz command.

9.5.1 Updating graph attributes

Dot.**style**(**attr*)

Sets the overall style (graph attributes) to the given attributes.

See *Valid Attributes* for more information about the attributes.

Dot.**node_style**(*node*, **attr*)

Sets the style for *node* to the given attributes.

This method will add *node* to the graph when it isn't already present.

See *Valid Attributes* for more information about the attributes.

Dot.**all_node_style**(**attr*)

Replaces the current style for all nodes

altgraph.Dot.**edge_style**(*head*, *tail*, **attr*)

Sets the style of an edge to the given attributes. The edge will be added to the graph when it isn't already present, but *head* and *tail* must both be valid nodes.

See *Valid Attributes* for more information about the attributes.

9.5.2 Emitting output

Dot.**display**([*mode*])

Displays the current graph via dotty.

If the *mode* is "neato" the dot file is processed with the neato command before displaying.

This method won't return until the dotty command exits.

altgraph.Dot.**save_dot**(*filename*)
Saves the current graph representation into the given file.

Note: For backward compatibility reasons this method can also be called without an argument, it will then write the graph into a fixed filename (present in the attribute Graph.temp_dot).

This feature is deprecated and should not be used.

---

altgraph.Dot.**save_image**(*file_name*[, *file_type*[, *mode*]])
Saves the current graph representation as an image file. The output is written into a file whose basename is *file_name* and whose suffix is *file_type*.

The *file_type* specifies the type of file to write, the default is "gif".

If the *mode* is "neato" the dot file is processed with the neato command before displaying.

Note: For backward compatibility reasons this method can also be called without an argument, it will then write the graph with a fixed basename ("out").

This feature is deprecated and should not be used.

---

altgraph.Dot.**iterdot**()
Yields all lines of a graphviz input file (including line endings).

altgraph.Dot.**__iter__**()
Alias for the *iterdot()* method.

CHAPTER 10

Online Resources

- Sourcecode repository on GitHub
- The issue tracker

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G