

# Metodi NetworkX

GRAFO SEMPLICE → `nx.Graph()`

## Reporting nodes edges and neighbors

<code>Graph.nodes</code>	A NodeView of the Graph as <code>G.nodes</code> or <code>G.nodes()</code> .
<code>Graph.iter</code> ()	Iterate over the nodes.
<code>Graph.has_node</code> (n)	Returns True if the graph contains the node n.
<code>Graph.__contains__</code> (n)	Returns True if n is a node, False otherwise.
<code>Graph.edges</code>	An EdgeView of the Graph as <code>G.edges</code> or <code>G.edges()</code> .
<code>Graph.has_edge</code> (u, v)	Returns True if the edge (u, v) is in the graph.
<code>Graph.get_edge_data</code> (u, v[, default])	Returns the attribute dictionary associated with edge (u, v).
<code>Graph.neighbors</code> (n)	Returns an iterator over all neighbors of node n.
<code>Graph.adj</code>	Graph adjacency object holding the neighbors of each node.
<code>Graph.getitem</code> (n)	Returns a dict of neighbors of node n.
<code>Graph.adjacency</code> ()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
<code>Graph.nbunch_iter</code> ([nbunch])	Returns an iterator over nodes contained in nbunch that are also in the graph.

## Counting nodes edges and neighbors

<code>Graph.order</code> ()	Returns the number of nodes in the graph.
<code>Graph.number of nodes</code> ()	Returns the number of nodes in the graph.
<code>Graph.__len__</code> ()	Returns the number of nodes in the graph.
<code>Graph.degree</code>	A DegreeView for the Graph as <code>G.degree</code> or <code>G.degree()</code> .
<code>Graph.size</code> ([weight])	Returns the number of edges or total of all edge weights.
<code>Graph.number of edges</code> ([u, v])	Returns the number of edges between two nodes.

## GRAFO ORIENTATO → nx.DiGraph()

### Adding and removing nodes and edges #

<code>DiGraph.__init__</code> ((incoming_graph_data))	Initialize a graph with edges, name, or graph attributes.
<code>DiGraph.add_node</code> (node_for_adding, **attr)	Add a single node <code>node_for_adding</code> and update node attributes.
<code>DiGraph.add_nodes_from</code> (nodes_for_adding, **attr)	Add multiple nodes.
<code>DiGraph.remove_node</code> (n)	Remove node n.
<code>DiGraph.remove_nodes_from</code> (nodes)	Remove multiple nodes.
<code>DiGraph.add_edge</code> (u_of_edge, v_of_edge, **attr)	Add an edge between u and v.
<code>DiGraph.add_edges_from</code> (ebunch_to_add, **attr)	Add all the edges in ebunch_to_add.
<code>DiGraph.add_weighted_edges_from</code> (ebunch_to_add)	Add weighted edges in <code>ebunch_to_add</code> with specified weight attr
<code>DiGraph.remove_edge</code> (u, v)	Remove the edge between u and v.
<code>DiGraph.remove_edges_from</code> (ebunch)	Remove all edges specified in ebunch.
<code>DiGraph.update</code> ((edges, nodes))	Update the graph using nodes/edges/graphs as input.
<code>DiGraph.clear</code> ()	Remove all nodes and edges from the graph.
<code>DiGraph.clear_edges</code> ()	Remove all edges from the graph without altering nodes.

### Reporting nodes edges and neighbors

<code>DiGraph.nodes</code>	A NodeView of the Graph as G.nodes or G.nodes().	<code>DiGraph.neighbors</code> (n)	Returns an iterator over successor nodes of n.
<code>DiGraph.iter</code> ()	Iterate over the nodes.	<code>DiGraph.adj</code>	Graph adjacency object holding the neighbors of each node.
<code>DiGraph.has_node</code> (n)	Returns True if the graph contains the node n.	<code>DiGraph.getitem</code> (n)	Returns a dict of neighbors of node n.
<code>DiGraph.contains</code> (n)	Returns True if n is a node, False otherwise.	<code>DiGraph.successors</code> (n)	Returns an iterator over successor nodes of n.
<code>DiGraph.edges</code>	An OutEdgeView of the DiGraph as G.edges or G.edges().	<code>DiGraph.succ</code>	Graph adjacency object holding the successors of each node.
<code>DiGraph.out_edges</code>	An OutEdgeView of the DiGraph as G.edges or G.edges().	<code>DiGraph.predecessors</code> (n)	Returns an iterator over predecessor nodes of n.
<code>DiGraph.in_edges</code>	A view of the in edges of the graph as G.in_edges or G.in_edges().	<code>DiGraph.pred</code>	Graph adjacency object holding the predecessors of each node.
<code>DiGraph.has_edge</code> (u, v)	Returns True if the edge (u, v) is in the graph.	<code>DiGraph.adjacency</code> ()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
<code>DiGraph.get_edge_data</code> (u, v[, default])	Returns the attribute dictionary associated with edge (u, v).	<code>DiGraph.nbunch_iter</code> ((nbunch))	Returns an iterator over nodes contained in nbunch that are also in the graph.

### Counting nodes edges and neighbors #

<code>DiGraph.order</code> ()	Returns the number of nodes in the graph.
<code>DiGraph.number_of_nodes</code> ()	Returns the number of nodes in the graph.
<code>DiGraph.len</code> ()	Returns the number of nodes in the graph.
<code>DiGraph.degree</code>	A DegreeView for the Graph as G.degree or G.degree().
<code>DiGraph.in_degree</code>	An InDegreeView for (node, in_degree) or in_degree for single node.
<code>DiGraph.out_degree</code>	An OutDegreeView for (node, out_degree)
<code>DiGraph.size</code> ([weight])	Returns the number of edges or total of all edge weights.
<code>DiGraph.number_of_edges</code> ([u, v])	Returns the number of edges between two nodes.

## Connectivity

<code>is_connected</code> (G)	Returns True if the graph is connected, False otherwise.
<code>number_connected_components</code> (G)	Returns the number of connected components.
<code>connected_components</code> (G)	Generate connected components.
<code>node_connected_component</code> (G, n)	Returns the set of nodes in the component of graph containing node n.

## Strong connectivity

<code>is_strongly_connected</code> (G)	Test directed graph for strong connectivity.
<code>number_strongly_connected_components</code> (G)	Returns number of strongly connected components in graph.
<code>strongly_connected_components</code> (G)	Generate nodes in strongly connected components of graph.
<code>kosaraju_strongly_connected_components</code> (G[, ...])	Generate nodes in strongly connected components of graph.
<code>condensation</code> (G[, scc])	Returns the condensation of G.

## Weak connectivity

<code>is_weakly_connected</code> (G)	Test directed graph for weak connectivity.
<code>number_weakly_connected_components</code> (G)	Returns the number of weakly connected components in G.
<code>weakly_connected_components</code> (G)	Generate weakly connected components of G.

## Depth First Search

Basic algorithms for depth-first searching the nodes of a graph.

<code>dfs_edges</code> (G[, source, depth_limit, ...])	Iterate over edges in a depth-first-search (DFS).
<code>dfs_tree</code> (G[, source, depth_limit, ...])	Returns oriented tree constructed from a depth-first-search from source.
<code>dfs_predecessors</code> (G[, source, depth_limit, ...])	Returns dictionary of predecessors in depth-first-search from source.
<code>dfs_successors</code> (G[, source, depth_limit, ...])	Returns dictionary of successors in depth-first-search from source.
<code>dfs_preorder_nodes</code> (G[, source, depth_limit, ...])	Generate nodes in a depth-first-search pre-ordering starting at source.
<code>dfs_postorder_nodes</code> (G[, source, ...])	Generate nodes in a depth-first-search post-ordering starting at source.
<code>dfs_labeled_edges</code> (G[, source, depth_limit, ...])	Iterate over edges in a depth-first-search (DFS) labeled by type.

## Breadth First Search

Basic algorithms for breadth-first searching the nodes of a graph.

<code>bfs_edges</code> (G, source[, reverse, depth_limit, ...])	Iterate over edges in a breadth-first-search starting at source.
<code>bfs_layers</code> (G, sources)	Returns an iterator of all the layers in breadth-first search traversal.
<code>bfs_tree</code> (G, source[, reverse, depth_limit, ...])	Returns an oriented tree constructed from of a breadth-first-search starting at source.
<code>bfs_predecessors</code> (G, source[, depth_limit, ...])	Returns an iterator of predecessors in breadth-first-search from source.
<code>bfs_successors</code> (G, source[, depth_limit, ...])	Returns an iterator of successors in breadth-first-search from source.
<code>descendants_at_distance</code> (G, source, distance)	Returns all nodes at a fixed <code>distance</code> from <code>source</code> in <code>G</code> .
<code>generic_bfs_edges</code> (G, source[, neighbors, ...])	Iterate over edges in a breadth-first search.

METODI PER CICLI SU GRAFI

Cycles

<code>cycle_basis</code> (G[, root])	Returns a list of cycles which form a basis for cycles of G.
<code>simple_cycles</code> (G[, length_bound])	Find simple cycles (elementary circuits) of a graph.
<code>recursive_simple_cycles</code> (G)	Find simple cycles (elementary circuits) of a directed graph.
<code>find_cycle</code> (G[, source, orientation])	Returns a cycle found via depth-first traversal.
<code>minimum_cycle_basis</code> (G[, weight])	Returns a minimum weight cycle basis for G
<code>chordless_cycles</code> (G[, length_bound])	Find simple chordless cycles of a graph.
<code>girth</code> (G)	Returns the girth of the graph.

<code>dag_longest_path</code> (G[, weight, ...])	Returns the longest path in a directed acyclic graph (DAG).
<code>dag_longest_path_length</code> (G[, weight, ...])	Returns the longest path length in a DAG

CAMMINI EURELIANI

<code>is_eulerian</code> (G)	Returns True if and only if <code>G</code> is Eulerian.
<code>eulerian_circuit</code> (G[, source, keys])	Returns an iterator over the edges of an Eulerian circuit in <code>G</code> .
<code>eulerize</code> (G)	Transforms a graph into an Eulerian graph.
<code>is_semieulerian</code> (G)	Return True iff <code>G</code> is semi-Eulerian.
<code>has_eulerian_path</code> (G[, source])	Return True iff <code>G</code> has an Eulerian path.
<code>eulerian_path</code> (G[, source, keys])	Return an iterator over the edges of an Eulerian path in <code>G</code> .

OPERATORI SU GRAFI

Operators

Unary operations on graphs

<code>complement</code> (G)	Returns the graph complement of G.
<code>reverse</code> (G[, copy])	Returns the reverse directed graph of G.

Operations on graphs including union, intersection, difference.

<code>compose</code> (G, H)	Compose graph G with H by combining nodes and edges into a single graph.
<code>union</code> (G, H[, rename])	Combine graphs G and H.
<code>disjoint_union</code> (G, H)	Combine graphs G and H.
<code>intersection</code> (G, H)	Returns a new graph that contains only the nodes and the edges that exist in both G and H.
<code>difference</code> (G, H)	Returns a new graph that contains the edges that exist in G but not in H.
<code>symmetric_difference</code> (G, H)	Returns new graph with edges that exist in either G or H but not both.
<code>full_join</code> (G, H[, rename])	Returns the full join of graphs G and H.

Operations on many graphs.

<code>compose_all</code> (graphs)	Returns the composition of all graphs.
<code>union_all</code> (graphs[, rename])	Returns the union of all graphs.
<code>disjoint_union_all</code> (graphs)	Returns the disjoint union of all graphs.
<code>intersection_all</code> (graphs)	Returns a new graph that contains only the nodes and the edges that exist in all graphs.

# Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

<code>shortest_path</code> ( <i>G</i> , <i>source</i> , <i>target</i> , <i>weight</i> , ...))	Compute shortest paths in the graph.
<code>all_shortest_paths</code> ( <i>G</i> , <i>source</i> , <i>target</i> , ...))	Compute all shortest simple paths in the graph.
<code>all_pairs_all_shortest_paths</code> ( <i>G</i> , <i>weight</i> , <i>method</i> ))	Compute all shortest paths between all nodes.
<code>single_source_all_shortest_paths</code> ( <i>G</i> , <i>source</i> )	Compute all shortest simple paths from the given source in the graph.
<code>shortest_path_length</code> ( <i>G</i> , <i>source</i> , <i>target</i> , ...))	Compute shortest path lengths in the graph.
<code>average_shortest_path_length</code> ( <i>G</i> , <i>weight</i> , <i>method</i> ))	Returns the average shortest path length.
<code>has_path</code> ( <i>G</i> , <i>source</i> , <i>target</i> )	Returns <i>True</i> if <i>G</i> has a path from <i>source</i> to <i>target</i> .

## Shortest path algorithms for weighted graphs.

<code>dijkstra_predecessor_and_distance</code> (G, source)	Compute weighted shortest path length and predecessors.
<code>dijkstra_path</code> (G, source, target[, weight])	Returns the shortest weighted path from source to target in G.
<code>dijkstra_path_length</code> (G, source, target[, weight])	Returns the shortest weighted path length in G from source to target.
<code>single_source_dijkstra</code> (G, source[, target, ...])	Find shortest weighted paths and lengths from a source node.
<code>single_source_dijkstra_path</code> (G, source[, ...])	Find shortest weighted paths in G from a source node.
<code>single_source_dijkstra_path_length</code> (G, source)	Find shortest weighted path lengths in G from a source node.
<code>multi_source_dijkstra</code> (G, sources[, target, ...])	Find shortest weighted paths and lengths from a given set of source nodes.
<code>multi_source_dijkstra_path</code> (G, sources[, ...])	Find shortest weighted paths in G from a given set of source nodes.
<code>multi_source_dijkstra_path_length</code> (G, sources)	Find shortest weighted path lengths in G from a given set of source nodes.