Graphs with NetworkX

Programmazione Avanzata 2025-26

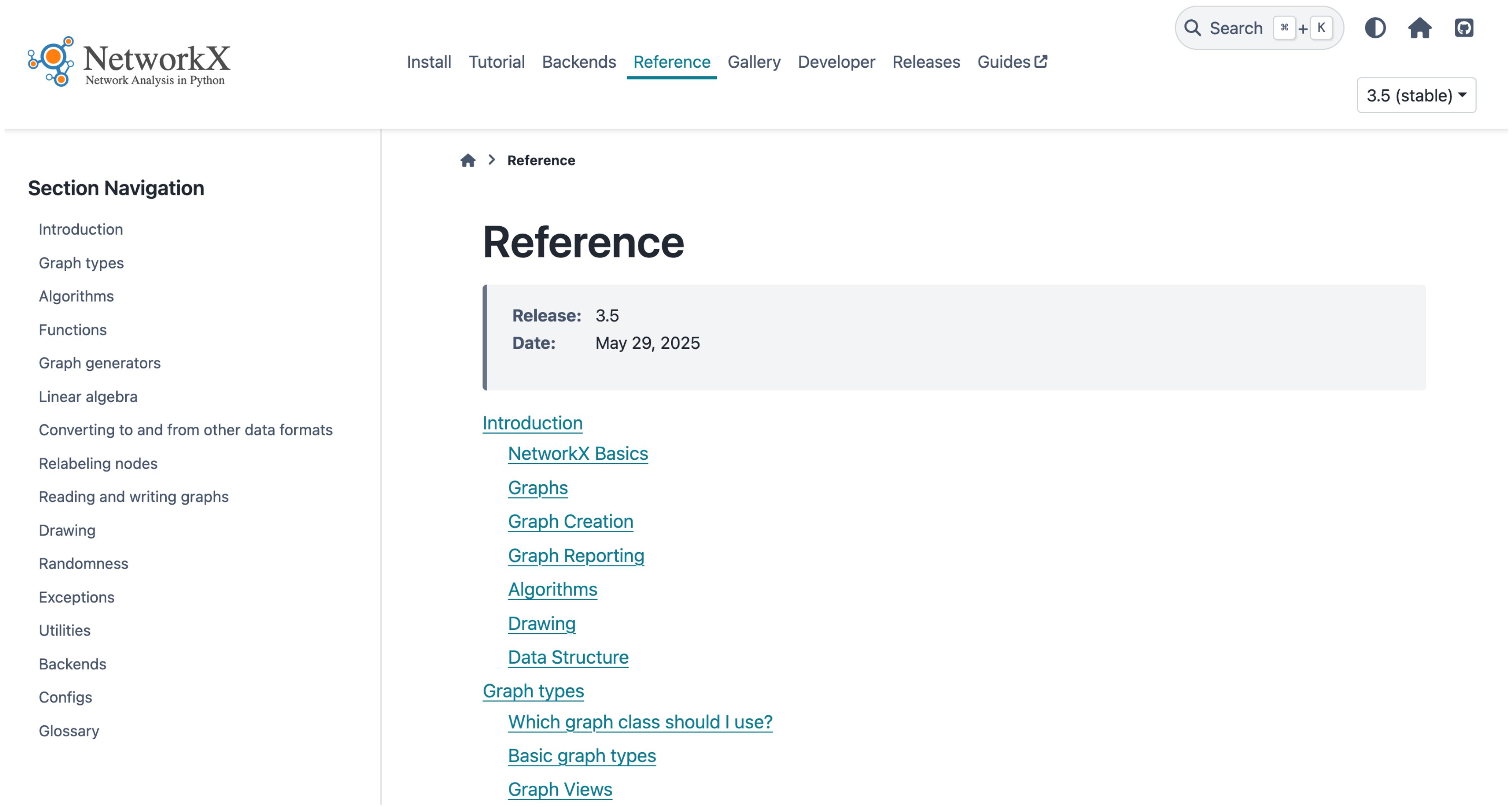
NetworkX

- Vast amounts of network data are being generated and collected
  - Sociology: web pages, mobile phones, social networks
  - Technology: Internet routers, vehicular flows, power grids
  - …
- How to analyse these networks?
  - Python + NetworkX
  - https://networkx.org

NetworkX

- "Python package for the creation, manipulation and study of the structure, dynamics and functions of complex networks."
  - Data structures for representing many types of data in the form of graphs
  - Nodes can be any Python object, edges can contain arbitrary data
  - Flexibility ideal for representing networks found in many different fields
  - Easy to install on multiple platforms
  - Online up-to-date documentation

Documentation



Section Navigation

- Introduction
- Graph types
- Algorithms
- Functions
- Graph generators
- Linear algebra
- Converting to and from other data formats
- Relabeling nodes
- Reading and writing graphs
- Drawing
- Randomness
- Exceptions
- Utilities
- Backends
- Configs
- Glossary

Reference

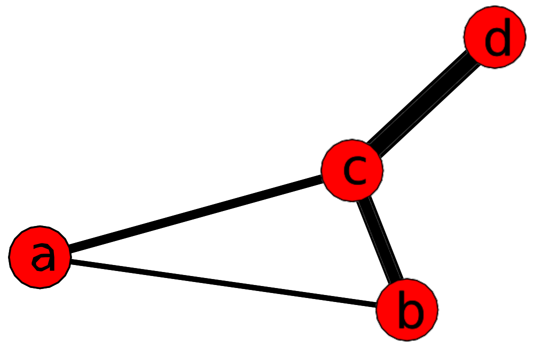
Release: 3.5
Date: May 29, 2025

Object model

- NetworkX defines no custom node objects or edge objects
  - Node-centric view of network
  - Nodes can be any hashable object, while edges are tuples with optional edge data (stored in dictionary)
  - Any Python object is allowed as edge data and it is assigned and stored in a Python dictionary (default empty)

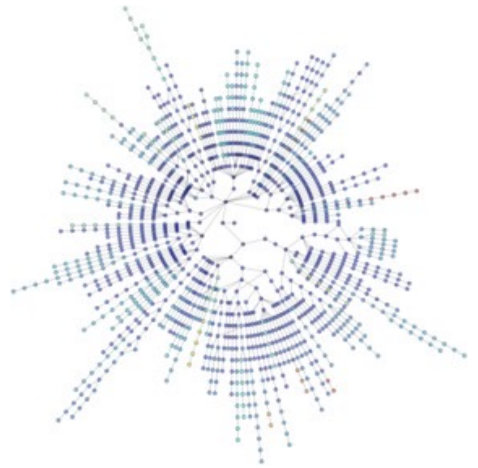
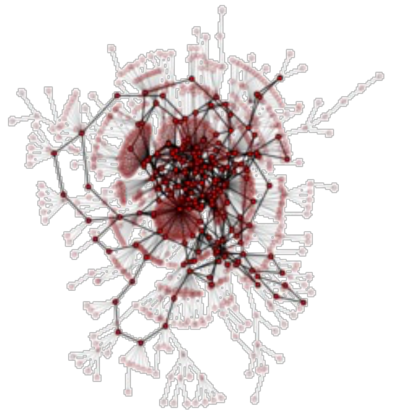
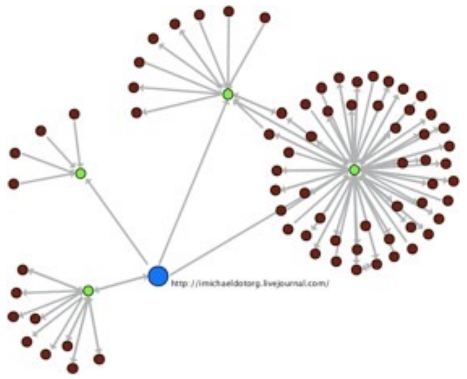
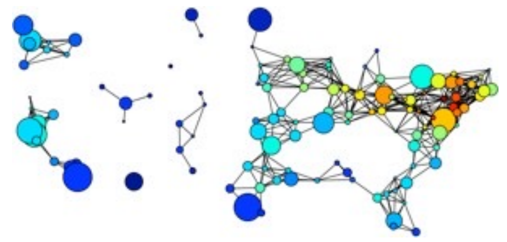
Example

```
import networkx as nx
g = nx.Graph()
g.add_edge("a","b",weight=1)
g.add_edge("b","c",weight=100)
g.add_edge("a","c",weight=1)
g.add_edge("c","d",weight=1)
print(nx.shortest_path(g,"b","d"))
print(nx.dijkstra_path(g,"b","d",weight='weight'))
```



Drawing and plotting

- It is possible to draw small graphs within NetworkX and to export network data and draw with other programs (i.e., GraphViz, matplotlib)

Getting started

- NetworkX supports many different graph types, like:
  - **`nx.Graph()`**, undirected
  - **`nx.DiGraph()`**, directed
  - **`nx.MultiGraph()`**, supports multiple edges between nodes
  - **`nx.MultiDiGraph()`**, directed multigraph
- Also provides implementation of notable graphs (like heawood)

Building a graph

- Nodes could be (almost) anything
  - Numbers, strings
  - Objects
  - Functions
  - Flet containers
  - …
- Edges connect nodes (even heterogeneous)
- Nodes and edges could have attributes

Example

```
import networkx as nx
import math
import flet as ft

g = nx.heawood_graph()
print(g.nodes, g.edges)
g.add_node(math.cos)
g.add_node(ft.Text("foo"))
g.add_edge("math.cos", 3)
print(g.nodes, g.edges)
```

Example

```
import networkx as nx
import math
import flet as ft

g = nx.Graph()
g.add_edge(1, 2) # default edge data = 1
g.add_edge(2, 3, weight=0.9) # specify edge data

g.add_edge('y', 'x', function=math.cos)
g.add_node(math.cos) # any hashable can be a node

elist = [(1, 2), (2, 3), (1, 4), (4, 2)]
g.add_edges_from(elist)
elist = [('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
g.add_weighted_edges_from(elist)
g.add_node(ft.Text("foo"))

print(g.nodes())
print(g.edges())
print(g.get_edge_data('a', 'b'))
```

Data structure

- A graph **`g`** is essentially a “dictionary of dictionaries of dictionaries”
- The keys of **`g`** are the nodes
- With **`g[n]`** one gets a dictionary where keys are all the nodes connected with node **`n`** (adjacency) and values are the edges parameters (like weight)

Example

```
import networkx as nx

g = nx.Graph()
g.add_edge(1, 2) # default edge data = 1
g.add_edge(2, 3, weight=0.9) # specify edge data

elist = [(1, 2, 1), (2, 3, 1), (1, 4, 1), (4, 2, 1),
         ('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0),
         ('c', 'd', 7.3)]

g.add_weighted_edges_from(elist)

print(g[2])

>>> {1: {'weight': 1}, 3: {'weight': 1}, 4: {'weight': 1}}
```

Data structure

- Common operations
  - **`g[u][v]`** yields the edge attributes
  - **`n in g`** tests if node **`n`** is in **`g`**
  - **`for n in g:`** iterates through the graph
  - **`for nbr in g[n]:`** iterates through the neighbors of **`n`**
  - **`g.nodes()`** and **`g.edges()`** provide corresponding data

Example

```
import networkx as nx

g = nx.Graph()
g.add_edge(1, 2) # default edge data = 1
g.add_edge(2, 3, weight=0.9) # specify edge data

elist = [(1, 2, 1), (2, 3, 1), (1, 4, 1), (4, 2, 1),
         ('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
g.add_weighted_edges_from(elist)
g.add_edge(2, 5, arbitraryAttr="foo")

print(g[2])
print("---------")
print(g['a']['b'])
print("---------")
print('e' in g)
print("---------")
for n in g:
    print(n)
print("---------")
for nbr in g[2]:
    print(nbr)
print("---------")
print(g[2][5]['arbitraryAttr'])
```

Directed and multi graphs

- Graphs can be directed, therefore differentiating neighbors in predecessors and successors
- Data structure for direct graphs is only slightly more complex: two dictionaries, one for successors and one for predecessors
- In multigraphs, two nodes can have more than one edge

Example

```
import networkx as nx

dg = nx.DiGraph()
dg.add_weighted_edges_from([(1, 4, 0.5), (3, 1, 0.75)])

print([s for s in dg.successors(1)])
print([p for p in dg.predecessors(1)])


mg = nx.MultiGraph()

mg.add_weighted_edges_from([(1, 2, .5), (1, 2, .75), (2, 3,
.5)])

print(mg[1][2])
```

Graph operators

- Classic graph operations
  - **`subgraph(G, nbunch)`**, induces subgraph of **`G`** on nodes in **`nbunch`**
  - **`union(G1,G2)`**, graph union
  - **`disjoint_union(G1,G2)`**, graph union assum. all nodes are diff.
  - **`compose(G1,G2)`**, combines graphs identif. nodes common to both
  - **`complement(G)`**, graph complement
  - **`create_empty_copy(G)`**, returns an empty copy of the same graph class
  - **`convert_to_undirected(G)`**, returns an undirected representation of **`G`**
  - **`convert_to_directed(G)`**, returns a directed representation of **`G`**