ODSCWest23_ex1

October 25, 2023

[114]:
```
# First example notebook for the ODSC West 2023 Workship:
# https://odsc.com/speakers/
 ↪using-graphs-for-large-feature-engineering-pipelines/
```

[188]:
```
import datetime

import pandas as pd

from graphreduce.node import GraphReduceNode
from graphreduce.graph_reduce import GraphReduce
from graphreduce.enum import ComputeLayerEnum, PeriodUnit
```

1 defining a node

[189]:
```
class CustomerNode(GraphReduceNode):
    def do_annotate(self):
        self.df[self.colabbr('name_length')] = self.df[self.colabbr('name')].
 ↪apply(lambda x: len(x))

    def do_filters(self):
        pass

    def do_normalize(self):
        pass

    def do_post_join_annotate(self):
        pass

    def do_reduce(self, reduce_key, *args, **kwargs):
        pass

    def do_labels(self, reduce_key, *args, **kwargs):
        pass
```

[212]:
```
class OrderNode(GraphReduceNode):
    def do_annotate(self):
```

```
        pass

    def do_filters(self):
        pass

    def do_normalize(self):
        pass

    def do_post_join_annotate(self):
        pass

    def do_reduce(self, reduce_key):
        return self.prep_for_features().groupby(self.colabbr(reduce_key)).agg(
            **{
                self.colabbr(f'{self.pk}_count') : pd.NamedAgg(column=self.
↪colabbr(self.pk), aggfunc='count'),
                self.colabbr(f'amount_sum'): pd.NamedAgg(column=self.
↪colabbr('amount'), aggfunc='sum')
            }
        ).reset_index()

    def do_labels(self, reduce_key):
        return self.prep_for_labels().groupby(self.colabbr(reduce_key)).agg(
            **{
                self.colabbr(f'{self.pk}_had_order') : pd.NamedAgg(column=self.
↪colabbr(self.pk), aggfunc='count')
            }
        ).reset_index()
```

2 Instantiate the node

[191]:
```
cust = CustomerNode(
    pk='id',
    prefix='cust',
    fpath='dat/cust.csv',
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
)
```

```
2023-10-25 18:25:50 [warning  ] no `date_key` set for <GraphReduceNode:
fpath=dat/cust.csv fmt=csv>
```

[192]:
```
cust.do_data()
```

[193]:
```
cust.df
```

[193]:

	cust_id	cust_name
0	1	wes
1	2	john

[ ]:

```
```

[ ]:

```
```

[194]:

```
order = OrderNode(
    pk='id',
    prefix='order',
    fpath='dat/orders.csv',
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
)
```

```
2023-10-25 18:25:52 [warning  ] no `date_key` set for <GraphReduceNode:
fpath=dat/orders.csv fmt=csv>
```

[195]:

```
order.do_data()
```

[196]:

```
order.df
```

[196]:

	order_id	order_customer_id	order_ts	order_amount
0	1	1	2023-05-12	10.0
1	2	1	2023-06-01	11.5
2	3	2	2023-01-01	100.0
3	4	2	2022-08-05	150.0
4	5	1	2023-07-01	325.0
5	6	2	2023-07-02	23.0
6	7	1	2023-07-14	12000.0

3 Run operations

[197]:

```
order.do_annotate()
```

[198]:

```
# pre-annotate
cust.df
```

[198]:

	cust_id	cust_name
0	1	wes
1	2	john

[199]:

```
cust.do_annotate()
```

[200]:

```
cust.df
```

[200]:

	cust_id	cust_name	cust_name_length
0	1	wes	3
1	2	john	4

[ ]:

```

```

4 Handling time

[201]:

```
len(order.df)
```

[201]: 7

[202]:

```
len(order.prep_for_features())
```

[202]: 7

[213]:

```
# we didn't provide a date key or date information
order = OrderNode(
    pk='id',
    prefix='order',
    fpath='dat/orders.csv',
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
    compute_period_val=365,
    compute_period_unit=PeriodUnit.day,
    label_period_val=30,
    label_period_unit=PeriodUnit.day,
    cut_date=datetime.datetime(2023, 6, 1),
    date_key='ts'
)
```

[214]:

```
order.do_data()
```

[215]:

```
print(len(order.df))
```

7

[216]:

```
print(len(order.prep_for_features()))
```

4

[217]:

```
order.prep_for_features()
```

[217]:

	order_id	order_customer_id	order_ts	order_amount
0	1	1	2023-05-12	10.0
1	2	1	2023-06-01	11.5

```
2           3                   2  2023-01-01         100.0
3           4                   2  2022-08-05         150.0
```

[ ]:

```

```

[218]:

```
order.prep_for_labels()
```

[218]:

	order_id	order_customer_id	order_ts	order_amount
4	5	1	2023-07-01	325.0

[ ]:

```

```

5 Adding operations to a node.

[209]:

```
order.do_reduce('customer_id')
```

[209]:

	order_customer_id	order_id_count
0	1	2
1	2	2

[219]:

```
# let's add a sum of the order amount
order.do_reduce('customer_id')
```

[219]:

	order_customer_id	order_id_count	order_amount_sum
0	1	2	21.5
1	2	2	250.0

[ ]:

```

```

[ ]:

```

```

6 Constructing a graph.

[220]:

```
help(GraphReduce)
```

```
Help on class GraphReduce in module graphreduce.graph_reduce:

class GraphReduce(networkx.classes.digraph.DiGraph)
 |  GraphReduce(name: str = 'graph_reduce', parent_node:
Optional[graphreduce.node.GraphReduceNode] = None, fmt: str = 'parquet',
compute_layer: graphreduce.enum.ComputeLayerEnum = None, cut_date:
datetime.datetime = datetime.datetime(2023, 10, 25, 16, 41, 54, 934704),
compute_period_val: Union[int, float] = 365, compute_period_unit:
graphreduce.enum.PeriodUnit = <PeriodUnit.day: 'day'>, has_labels: bool = False,
label_period_val: Union[int, float, NoneType] = None, label_period_unit:
```

```
Optional[graphreduce.enum.PeriodUnit] = None, spark_sqlctx:
pyspark.sql.context.SQLContext = None, feature_function: Optional[str] = None,
dynamic_propagation: bool = False, type_func_map: Dict[str, List[str]] =
{'int64': ['min', 'max', 'sum'], 'str': ['first'], 'object': ['first'],
'float64': ['min', 'max', 'sum'], 'bool': ['first'], 'datetime64': ['first']},
storage_client: Optional[graphreduce.storage.StorageClient] = None, *args,
**kwargs)
 |
 |  Method resolution order:
 |      GraphReduce
 |      networkx.classes.digraph.DiGraph
 |      networkx.classes.graph.Graph
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, name: str = 'graph_reduce', parent_node:
Optional[graphreduce.node.GraphReduceNode] = None, fmt: str = 'parquet',
compute_layer: graphreduce.enum.ComputeLayerEnum = None, cut_date:
datetime.datetime = datetime.datetime(2023, 10, 25, 16, 41, 54, 934704),
compute_period_val: Union[int, float] = 365, compute_period_unit:
graphreduce.enum.PeriodUnit = <PeriodUnit.day: 'day'>, has_labels: bool = False,
label_period_val: Union[int, float, NoneType] = None, label_period_unit:
Optional[graphreduce.enum.PeriodUnit] = None, spark_sqlctx:
pyspark.sql.context.SQLContext = None, feature_function: Optional[str] = None,
dynamic_propagation: bool = False, type_func_map: Dict[str, List[str]] =
{'int64': ['min', 'max', 'sum'], 'str': ['first'], 'object': ['first'],
'float64': ['min', 'max', 'sum'], 'bool': ['first'], 'datetime64': ['first']},
storage_client: Optional[graphreduce.storage.StorageClient] = None, *args,
**kwargs)
 |      Constructor for GraphReduce
 |
 |      Args:
 |          name : the name of the graph reduce
 |          parent_node : parent-most node in the graph, if doing reductions the
granularity to which to reduce the data
 |          fmt : the format of the dataset
 |          compute_layer : compute layer to use (e.g., spark)
 |          cut_date : the date to cut off history
 |          compute_period_val : the amount of time to consider during the
compute job
 |          compute_period_unit : the unit for the compute period value (e.g.,
day)
 |          has_labels : whether or not the compute job computes labels, when
True `prep_for_labels()` and `compute_labels` will be called
 |          label_period_val : amount of time to consider when computing labels
 |          label_period_unit : the unit for the label period value (e.g., day)
 |          spark_sqlctx : if compute layer is spark this must be passed
```

```
 |          feature_function : optional custom feature function
 |          dynamic_propagation : optional to dynamically propagate children
data upward, useful for very large compute graphs
 |          type_func_match : optional mapping from type to a list of functions
(e.g., {'int' : ['min', 'max', 'sum'], 'str' : ['first']})
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  __str__(self)
 |      Returns a short summary of the graph.
 |
 |      Returns
 |      -------
 |      info : string
 |          Graph information as provided by `nx.info`
 |
 |      Examples
 |      --------
 |      >>> G = nx.Graph(name="foo")
 |      >>> str(G)
 |      "Graph named 'foo' with 0 nodes and 0 edges"
 |
 |      >>> G = nx.path_graph(3)
 |      >>> str(G)
 |      'Graph with 3 nodes and 2 edges'
 |
 |  add_entity_edge(self, parent_node: graphreduce.node.GraphReduceNode,
relation_node: graphreduce.node.GraphReduceNode, parent_key: str, relation_key:
str, relation_type: str = 'parent_child', reduce: bool = True)
 |      Add an entity relation
 |
 |  assign_parent(self, parent_node: graphreduce.node.GraphReduceNode)
 |      Assign the parent-most node in the graph
 |
 |  depth_first_generator(self)
 |      Depth-first traversal over the edges
 |
 |  do_transformations(self)
 |      Perform all graph transformations
 |      1) hydrate graph
 |      2) check for duplicate prefixes
 |      3) filter data
 |      4) clip anomalies
 |      5) annotate data
 |      6) depth-first edge traversal to: aggregate / reduce features and labels
 |      6a) optional alternative feature_function mapping
 |      6b) join back to parent edge
```

```
 |      6c) post-join annotations if any
 |      7) repeat step 6 on all edges up the hierarchy
 |
 |  get_children(self, node: graphreduce.node.GraphReduceNode) ->
List[graphreduce.node.GraphReduceNode]
 |      Get the children of a given node
 |
 |  hydrate_graph_attrs(self, attrs=['cut_date', 'compute_period_val',
'compute_period_unit', 'has_labels', 'label_period_val', 'label_period_unit',
'compute_layer', 'feature_function', 'spark_sqlctx', '_storage_client'])
 |      Hydrate the nodes in the graph with parent
 |      attributes in `attrs`
 |
 |  hydrate_graph_data(self)
 |      Hydrate the nodes in the graph with their data
 |
 |  join(self, parent_node: graphreduce.node.GraphReduceNode, relation_node:
graphreduce.node.GraphReduceNode, relation_df=None)
 |      Join the child or peer nnode to the parent node
 |
 |      Optionally pass the `child_df` directly
 |
 |  plot_graph(self, fname: str = 'graph.html')
 |      Plot the graph
 |
 |      Args
 |          fname : file name to save the graph to - should be .html
 |          notebook : whether or not to render in notebook
 |
 |  prefix_uniqueness(self)
 |      Identify children with duplicate prefixes, if any
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  parent
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from networkx.classes.digraph.DiGraph:
 |
 |  add_edge(self, u_of_edge, v_of_edge, **attr)
 |      Add an edge between u and v.
 |
 |      The nodes u and v will be automatically added if they are
 |      not already in the graph.
 |
 |      Edge attributes can be specified with keywords or by directly
 |      accessing the edge's attribute dictionary. See examples below.
```

```
 |
 |      Parameters
 |      ----------
 |      u_of_edge, v_of_edge : nodes
 |          Nodes can be, for example, strings or numbers.
 |          Nodes must be hashable (and not None) Python objects.
 |      attr : keyword arguments, optional
 |          Edge data (or labels or objects) can be assigned using
 |          keyword arguments.
 |
 |      See Also
 |      --------
 |      add_edges_from : add a collection of edges
 |
 |      Notes
 |      -----
 |      Adding an edge that already exists updates the edge data.
 |
 |      Many NetworkX algorithms designed for weighted graphs use
 |      an edge attribute (by default `weight`) to hold a numerical value.
 |
 |      Examples
 |      --------
 |      The following all add the edge e=(1, 2) to graph G:
 |
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> e = (1, 2)
 |      >>> G.add_edge(1, 2)  # explicit two-node form
 |      >>> G.add_edge(*e)  # single edge as tuple of two nodes
 |      >>> G.add_edges_from([(1, 2)])  # add edges from iterable container
 |
 |      Associate data to edges using keywords:
 |
 |      >>> G.add_edge(1, 2, weight=3)
 |      >>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
 |
 |      For non-string attribute keys, use subscript notation.
 |
 |      >>> G.add_edge(1, 2)
 |      >>> G[1][2].update({0: 5})
 |      >>> G.edges[1, 2].update({0: 5})
 |
 |  add_edges_from(self, ebunch_to_add, **attr)
 |      Add all the edges in ebunch_to_add.
 |
 |      Parameters
 |      ----------
 |      ebunch_to_add : container of edges
```

```
|           Each edge given in the container will be added to the
|           graph. The edges must be given as 2-tuples (u, v) or
|           3-tuples (u, v, d) where d is a dictionary containing edge data.
|       attr : keyword arguments, optional
|           Edge data (or labels or objects) can be assigned using
|           keyword arguments.
|
|       See Also
|       --------
|       add_edge : add a single edge
|       add_weighted_edges_from : convenient way to add weighted edges
|
|       Notes
|       -----
|       Adding the same edge twice has no effect but any edge data
|       will be updated when each duplicate edge is added.
|
|       Edge attributes specified in an ebunch take precedence over
|       attributes specified via keyword arguments.
|
|       Examples
|       --------
|       >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
|       >>> G.add_edges_from([(0, 1), (1, 2)])  # using a list of edge tuples
|       >>> e = zip(range(0, 3), range(1, 4))
|       >>> G.add_edges_from(e)  # Add the path graph 0-1-2-3
|
|       Associate data to edges
|
|       >>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
|       >>> G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
|
|   add_node(self, node_for_adding, **attr)
|       Add a single node `node_for_adding` and update node attributes.
|
|       Parameters
|       ----------
|       node_for_adding : node
|           A node can be any hashable Python object except None.
|       attr : keyword arguments, optional
|           Set or change node attributes using key=value.
|
|       See Also
|       --------
|       add_nodes_from
|
|       Examples
|       --------
```

```
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_node(1)
 |      >>> G.add_node("Hello")
 |      >>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
 |      >>> G.add_node(K3)
 |      >>> G.number_of_nodes()
 |      3
 |
 |      Use keywords set/change node attributes:
 |
 |      >>> G.add_node(1, size=10)
 |      >>> G.add_node(3, weight=0.4, UTM=("13S", 382871, 3972649))
 |
 |      Notes
 |      -----
 |      A hashable object is one that can be used as a key in a Python
 |      dictionary. This includes strings, numbers, tuples of strings
 |      and numbers, etc.
 |
 |      On many platforms hashable items also include mutables such as
 |      NetworkX Graphs, though one should be careful that the hash
 |      doesn't change on mutables.
 |
 |  add_nodes_from(self, nodes_for_adding, **attr)
 |      Add multiple nodes.
 |
 |      Parameters
 |      ----------
 |      nodes_for_adding : iterable container
 |          A container of nodes (list, dict, set, etc.).
 |          OR
 |          A container of (node, attribute dict) tuples.
 |          Node attributes are updated using the attribute dict.
 |      attr : keyword arguments, optional (default= no attributes)
 |          Update attributes for all nodes in nodes.
 |          Node attributes specified in nodes as a tuple take
 |          precedence over attributes specified via keyword arguments.
 |
 |      See Also
 |      --------
 |      add_node
 |
 |      Examples
 |      --------
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_nodes_from("Hello")
 |      >>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
 |      >>> G.add_nodes_from(K3)
```

```
|      >>> sorted(G.nodes(), key=str)
|      [0, 1, 2, 'H', 'e', 'l', 'o']
|
|      Use keywords to update specific node attributes for every node.
|
|      >>> G.add_nodes_from([1, 2], size=10)
|      >>> G.add_nodes_from([3, 4], weight=0.4)
|
|      Use (node, attrdict) tuples to update attributes for specific nodes.
|
|      >>> G.add_nodes_from([(1, dict(size=11)), (2, {"color": "blue"})])
|      >>> G.nodes[1]["size"]
|      11
|      >>> H = nx.Graph()
|      >>> H.add_nodes_from(G.nodes(data=True))
|      >>> H.nodes[1]["size"]
|      11
|
|  adj = <functools.cached_property object>
|      Graph adjacency object holding the neighbors of each node.
|
|      This object is a read-only dict-like structure with node keys
|      and neighbor-dict values.  The neighbor-dict is keyed by neighbor
|      to the edge-data-dict.  So `G.adj[3][2]['color'] = 'blue'` sets
|      the color of the edge `(3, 2)` to `"blue"`.
|
|      Iterating over G.adj behaves like a dict. Useful idioms include
|      `for nbr, datadict in G.adj[n].items():`.
|
|      The neighbor information is also provided by subscripting the graph.
|      So `for nbr, foovalue in G[node].data('foo', default=1):` works.
|
|      For directed graphs, `G.adj` holds outgoing (successor) info.
|
|  clear(self)
|      Remove all nodes and edges from the graph.
|
|      This also removes the name, and all graph, node, and edge attributes.
|
|      Examples
|      --------
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> G.clear()
|      >>> list(G.nodes)
|      []
|      >>> list(G.edges)
|      []
|
```

```
 |  clear_edges(self)
 |      Remove all edges from the graph without altering nodes.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.clear_edges()
 |      >>> list(G.nodes)
 |      [0, 1, 2, 3]
 |      >>> list(G.edges)
 |      []
 |
 |  degree = <functools.cached_property object>
 |      A DegreeView for the Graph as G.degree or G.degree().
 |
 |      The node degree is the number of edges adjacent to the node.
 |      The weighted node degree is the sum of the edge weights for
 |      edges incident to that node.
 |
 |      This object provides an iterator for (node, degree) as well as
 |      lookup for the degree for a single node.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges incident to these nodes.
 |
 |      weight : string or None, optional (default=None)
 |         The name of an edge attribute that holds the numerical value used
 |         as a weight.  If None, then each edge has weight 1.
 |         The degree is the sum of the edge weights adjacent to the node.
 |
 |      Returns
 |      -------
 |      DiDegreeView or int
 |          If multiple nodes are requested (the default), returns a
`DiDegreeView`
 |          mapping nodes to their degree.
 |          If a single node is requested, returns the degree of the node as an
integer.
 |
 |      See Also
 |      --------
 |      in_degree, out_degree
 |
 |      Examples
 |      --------
 |      >>> G = nx.DiGraph()  # or MultiDiGraph
```

```
 |      >>> nx.add_path(G, [0, 1, 2, 3])
 |      >>> G.degree(0)  # node 0 with degree 1
 |      1
 |      >>> list(G.degree([0, 1, 2]))
 |      [(0, 1), (1, 2), (2, 2)]
 |
 |  edges = <functools.cached_property object>
 |      An OutEdgeView of the DiGraph as G.edges or G.edges().
 |
 |      edges(self, nbunch=None, data=False, default=None)
 |
 |      The OutEdgeView provides set-like operations on the edge-tuples
 |      as well as edge attribute lookup. When called, it also provides
 |      an EdgeDataView object which allows control of access to edge
 |      attributes (but does not provide set-like operations).
 |      Hence, `G.edges[u, v]['color']` provides the value of the color
 |      attribute for edge `(u, v)` while
 |      `for (u, v, c) in G.edges.data('color', default='red'):`
 |      iterates through all the edges yielding the color attribute
 |      with default `'red'` if no color attribute exists.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges from these nodes.
 |      data : string or bool, optional (default=False)
 |          The edge attribute returned in 3-tuple (u, v, ddict[data]).
 |          If True, return edge attribute dict in 3-tuple (u, v, ddict).
 |          If False, return 2-tuple (u, v).
 |      default : value, optional (default=None)
 |          Value used for edges that don't have the requested attribute.
 |          Only relevant if data is not True or False.
 |
 |      Returns
 |      -------
 |      edges : OutEdgeView
 |          A view of edge attributes, usually it iterates over (u, v)
 |          or (u, v, d) tuples of edges, but can also be used for
 |          attribute lookup as `edges[u, v]['foo']`.
 |
 |      See Also
 |      --------
 |      in_edges, out_edges
 |
 |      Notes
 |      -----
 |      Nodes in nbunch that are not in the graph will be (quietly) ignored.
 |      For directed graphs this returns the out-edges.
```

```
 |
 |      Examples
 |      --------
 |      >>> G = nx.DiGraph()  # or MultiDiGraph, etc
 |      >>> nx.add_path(G, [0, 1, 2])
 |      >>> G.add_edge(2, 3, weight=5)
 |      >>> [e for e in G.edges]
 |      [(0, 1), (1, 2), (2, 3)]
 |      >>> G.edges.data()  # default data is {} (empty dict)
 |      OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
 |      >>> G.edges.data("weight", default=1)
 |      OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
 |      >>> G.edges([0, 2])  # only edges originating from these nodes
 |      OutEdgeDataView([(0, 1), (2, 3)])
 |      >>> G.edges(0)  # only edges from node 0
 |      OutEdgeDataView([(0, 1)])
 |
 |  has_predecessor(self, u, v)
 |      Returns True if node u has predecessor v.
 |
 |      This is true if graph has the edge u<-v.
 |
 |  has_successor(self, u, v)
 |      Returns True if node u has successor v.
 |
 |      This is true if graph has the edge u->v.
 |
 |  in_degree = <functools.cached_property object>
 |      An InDegreeView for (node, in_degree) or in_degree for single node.
 |
 |      The node in_degree is the number of edges pointing to the node.
 |      The weighted node degree is the sum of the edge weights for
 |      edges incident to that node.
 |
 |      This object provides an iteration over (node, in_degree) as well as
 |      lookup for the degree for a single node.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges incident to these nodes.
 |
 |      weight : string or None, optional (default=None)
 |         The name of an edge attribute that holds the numerical value used
 |         as a weight.  If None, then each edge has weight 1.
 |         The degree is the sum of the edge weights adjacent to the node.
 |
 |      Returns
```

```
 |      -------
 |      If a single node is requested
 |      deg : int
 |          In-degree of the node
 |
 |      OR if multiple nodes are requested
 |      nd_iter : iterator
 |          The iterator returns two-tuples of (node, in-degree).
 |
 |      See Also
 |      --------
 |      degree, out_degree
 |
 |      Examples
 |      --------
 |      >>> G = nx.DiGraph()
 |      >>> nx.add_path(G, [0, 1, 2, 3])
 |      >>> G.in_degree(0)  # node 0 with degree 0
 |      0
 |      >>> list(G.in_degree([0, 1, 2]))
 |      [(0, 0), (1, 1), (2, 1)]
 |
 |  in_edges = <functools.cached_property object>
 |      An InEdgeView of the Graph as G.in_edges or G.in_edges().
 |
 |      in_edges(self, nbunch=None, data=False, default=None):
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges incident to these nodes.
 |      data : string or bool, optional (default=False)
 |          The edge attribute returned in 3-tuple (u, v, ddict[data]).
 |          If True, return edge attribute dict in 3-tuple (u, v, ddict).
 |          If False, return 2-tuple (u, v).
 |      default : value, optional (default=None)
 |          Value used for edges that don't have the requested attribute.
 |          Only relevant if data is not True or False.
 |
 |      Returns
 |      -------
 |      in_edges : InEdgeView
 |          A view of edge attributes, usually it iterates over (u, v)
 |          or (u, v, d) tuples of edges, but can also be used for
 |          attribute lookup as `edges[u, v]['foo']`.
 |
 |      See Also
 |      --------
```

```
|      edges
|
|  is_directed(self)
|      Returns True if graph is directed, False otherwise.
|
|  is_multigraph(self)
|      Returns True if graph is a multigraph, False otherwise.
|
|  neighbors = successors(self, n)
|
|  out_degree = <functools.cached_property object>
|      An OutDegreeView for (node, out_degree)
|
|      The node out_degree is the number of edges pointing out of the node.
|      The weighted node degree is the sum of the edge weights for
|      edges incident to that node.
|
|      This object provides an iterator over (node, out_degree) as well as
|      lookup for the degree for a single node.
|
|      Parameters
|      ----------
|      nbunch : single node, container, or all nodes (default= all nodes)
|          The view will only report edges incident to these nodes.
|
|      weight : string or None, optional (default=None)
|         The name of an edge attribute that holds the numerical value used
|         as a weight.  If None, then each edge has weight 1.
|         The degree is the sum of the edge weights adjacent to the node.
|
|      Returns
|      -------
|      If a single node is requested
|      deg : int
|          Out-degree of the node
|
|      OR if multiple nodes are requested
|      nd_iter : iterator
|          The iterator returns two-tuples of (node, out-degree).
|
|      See Also
|      --------
|      degree, in_degree
|
|      Examples
|      --------
|      >>> G = nx.DiGraph()
|      >>> nx.add_path(G, [0, 1, 2, 3])
```

```
 |      >>> G.out_degree(0)  # node 0 with degree 1
 |      1
 |      >>> list(G.out_degree([0, 1, 2]))
 |      [(0, 1), (1, 1), (2, 1)]
 |
 |  out_edges = <functools.cached_property object>
 |      An OutEdgeView of the DiGraph as G.edges or G.edges().
 |
 |      edges(self, nbunch=None, data=False, default=None)
 |
 |      The OutEdgeView provides set-like operations on the edge-tuples
 |      as well as edge attribute lookup. When called, it also provides
 |      an EdgeDataView object which allows control of access to edge
 |      attributes (but does not provide set-like operations).
 |      Hence, `G.edges[u, v]['color']` provides the value of the color
 |      attribute for edge `(u, v)` while
 |      `for (u, v, c) in G.edges.data('color', default='red'):`
 |      iterates through all the edges yielding the color attribute
 |      with default `'red'` if no color attribute exists.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges from these nodes.
 |      data : string or bool, optional (default=False)
 |          The edge attribute returned in 3-tuple (u, v, ddict[data]).
 |          If True, return edge attribute dict in 3-tuple (u, v, ddict).
 |          If False, return 2-tuple (u, v).
 |      default : value, optional (default=None)
 |          Value used for edges that don't have the requested attribute.
 |          Only relevant if data is not True or False.
 |
 |      Returns
 |      -------
 |      edges : OutEdgeView
 |          A view of edge attributes, usually it iterates over (u, v)
 |          or (u, v, d) tuples of edges, but can also be used for
 |          attribute lookup as `edges[u, v]['foo']`.
 |
 |      See Also
 |      --------
 |      in_edges, out_edges
 |
 |      Notes
 |      -----
 |      Nodes in nbunch that are not in the graph will be (quietly) ignored.
 |      For directed graphs this returns the out-edges.
 |
```

```
 |      Examples
 |      --------
 |      >>> G = nx.DiGraph()  # or MultiDiGraph, etc
 |      >>> nx.add_path(G, [0, 1, 2])
 |      >>> G.add_edge(2, 3, weight=5)
 |      >>> [e for e in G.edges]
 |      [(0, 1), (1, 2), (2, 3)]
 |      >>> G.edges.data()  # default data is {} (empty dict)
 |      OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
 |      >>> G.edges.data("weight", default=1)
 |      OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
 |      >>> G.edges([0, 2])  # only edges originating from these nodes
 |      OutEdgeDataView([(0, 1), (2, 3)])
 |      >>> G.edges(0)  # only edges from node 0
 |      OutEdgeDataView([(0, 1)])
 |
 |  pred = <functools.cached_property object>
 |      Graph adjacency object holding the predecessors of each node.
 |
 |      This object is a read-only dict-like structure with node keys
 |      and neighbor-dict values.  The neighbor-dict is keyed by neighbor
 |      to the edge-data-dict.  So `G.pred[2][3]['color'] = 'blue'` sets
 |      the color of the edge `(3, 2)` to `"blue"`.
 |
 |      Iterating over G.pred behaves like a dict. Useful idioms include
 |      `for nbr, datadict in G.pred[n].items():`.  A data-view not provided
 |      by dicts also exists: `for nbr, foovalue in G.pred[node].data('foo'):`
 |      A default can be set via a `default` argument to the `data` method.
 |
 |  predecessors(self, n)
 |      Returns an iterator over predecessor nodes of n.
 |
 |      A predecessor of n is a node m such that there exists a directed
 |      edge from m to n.
 |
 |      Parameters
 |      ----------
 |      n : node
 |         A node in the graph
 |
 |      Raises
 |      ------
 |      NetworkXError
 |         If n is not in the graph.
 |
 |      See Also
 |      --------
 |      successors
```

```
|
|  remove_edge(self, u, v)
|      Remove the edge between u and v.
|
|      Parameters
|      ----------
|      u, v : nodes
|          Remove the edge between nodes u and v.
|
|      Raises
|      ------
|      NetworkXError
|          If there is not an edge between u and v.
|
|      See Also
|      --------
|      remove_edges_from : remove a collection of edges
|
|      Examples
|      --------
|      >>> G = nx.Graph()  # or DiGraph, etc
|      >>> nx.add_path(G, [0, 1, 2, 3])
|      >>> G.remove_edge(0, 1)
|      >>> e = (1, 2)
|      >>> G.remove_edge(*e)  # unpacks e from an edge tuple
|      >>> e = (2, 3, {"weight": 7})  # an edge with attribute data
|      >>> G.remove_edge(*e[:2])  # select first part of edge tuple
|
|  remove_edges_from(self, ebunch)
|      Remove all edges specified in ebunch.
|
|      Parameters
|      ----------
|      ebunch: list or container of edge tuples
|          Each edge given in the list or container will be removed
|          from the graph. The edges can be:
|
|              - 2-tuples (u, v) edge between u and v.
|              - 3-tuples (u, v, k) where k is ignored.
|
|      See Also
|      --------
|      remove_edge : remove a single edge
|
|      Notes
|      -----
|      Will fail silently if an edge in ebunch is not in the graph.
|
```

```
|      Examples
|      --------
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> ebunch = [(1, 2), (2, 3)]
|      >>> G.remove_edges_from(ebunch)
|
|  remove_node(self, n)
|      Remove node n.
|
|      Removes the node n and all adjacent edges.
|      Attempting to remove a non-existent node will raise an exception.
|
|      Parameters
|      ----------
|      n : node
|         A node in the graph
|
|      Raises
|      ------
|      NetworkXError
|         If n is not in the graph.
|
|      See Also
|      --------
|      remove_nodes_from
|
|      Examples
|      --------
|      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> list(G.edges)
|      [(0, 1), (1, 2)]
|      >>> G.remove_node(1)
|      >>> list(G.edges)
|      []
|
|  remove_nodes_from(self, nodes)
|      Remove multiple nodes.
|
|      Parameters
|      ----------
|      nodes : iterable container
|          A container of nodes (list, dict, set, etc.).  If a node
|          in the container is not in the graph it is silently ignored.
|
|      See Also
|      --------
|      remove_node
|
```

```
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> e = list(G.nodes)
 |      >>> e
 |      [0, 1, 2]
 |      >>> G.remove_nodes_from(e)
 |      >>> list(G.nodes)
 |      []
 |
 |  reverse(self, copy=True)
 |      Returns the reverse of the graph.
 |
 |      The reverse is a graph with the same nodes and edges
 |      but with the directions of the edges reversed.
 |
 |      Parameters
 |      ----------
 |      copy : bool optional (default=True)
 |          If True, return a new DiGraph holding the reversed edges.
 |          If False, the reverse graph is created using a view of
 |          the original graph.
 |
 |  succ = <functools.cached_property object>
 |      Graph adjacency object holding the successors of each node.
 |
 |      This object is a read-only dict-like structure with node keys
 |      and neighbor-dict values.  The neighbor-dict is keyed by neighbor
 |      to the edge-data-dict.  So `G.succ[3][2]['color'] = 'blue'` sets
 |      the color of the edge `(3, 2)` to `"blue"`.
 |
 |      Iterating over G.succ behaves like a dict. Useful idioms include
 |      `for nbr, datadict in G.succ[n].items():`.  A data-view not provided
 |      by dicts also exists: `for nbr, foovalue in G.succ[node].data('foo'):`
 |      and a default can be set via a `default` argument to the `data` method.
 |
 |      The neighbor information is also provided by subscripting the graph.
 |      So `for nbr, foovalue in G[node].data('foo', default=1):` works.
 |
 |      For directed graphs, `G.adj` is identical to `G.succ`.
 |
 |  successors(self, n)
 |      Returns an iterator over successor nodes of n.
 |
 |      A successor of n is a node m such that there exists a directed
 |      edge from n to m.
 |
 |      Parameters
```

```
 |      ----------
 |      n : node
 |         A node in the graph
 |
 |      Raises
 |      ------
 |      NetworkXError
 |         If n is not in the graph.
 |
 |      See Also
 |      --------
 |      predecessors
 |
 |      Notes
 |      -----
 |      neighbors() and successors() are the same.
 |
 |  to_undirected(self, reciprocal=False, as_view=False)
 |      Returns an undirected representation of the digraph.
 |
 |      Parameters
 |      ----------
 |      reciprocal : bool (optional)
 |        If True only keep edges that appear in both directions
 |        in the original digraph.
 |      as_view : bool (optional, default=False)
 |        If True return an undirected view of the original directed graph.
 |
 |      Returns
 |      -------
 |      G : Graph
 |          An undirected graph with the same name and nodes and
 |          with edge (u, v, data) if either (u, v, data) or (v, u, data)
 |          is in the digraph.  If both edges exist in digraph and
 |          their edge data is different, only one edge is created
 |          with an arbitrary choice of which edge data to use.
 |          You must check and correct for this manually if desired.
 |
 |      See Also
 |      --------
 |      Graph, copy, add_edge, add_edges_from
 |
 |      Notes
 |      -----
 |      If edges in both directions (u, v) and (v, u) exist in the
 |      graph, attributes for the new undirected edge will be a combination of
 |      the attributes of the directed edges.  The edge data is updated
 |      in the (arbitrary) order that the edges are encountered.  For
```

```
 |      more customized control of the edge attributes use add_edge().
 |
 |      This returns a "deepcopy" of the edge, node, and
 |      graph attributes which attempts to completely copy
 |      all of the data and references.
 |
 |      This is in contrast to the similar G=DiGraph(D) which returns a
 |      shallow copy of the data.
 |
 |      See the Python copy module for more information on shallow
 |      and deep copies, https://docs.python.org/3/library/copy.html.
 |
 |      Warning: If you have subclassed DiGraph to use dict-like objects
 |      in the data structure, those changes do not transfer to the
 |      Graph created by this method.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(2)  # or MultiGraph, etc
 |      >>> H = G.to_directed()
 |      >>> list(H.edges)
 |      [(0, 1), (1, 0)]
 |      >>> G2 = H.to_undirected()
 |      >>> list(G2.edges)
 |      [(0, 1)]
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from networkx.classes.graph.Graph:
 |
 |  __contains__(self, n)
 |      Returns True if n is a node, False otherwise. Use: 'n in G'.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> 1 in G
 |      True
 |
 |  __getitem__(self, n)
 |      Returns a dict of neighbors of node n.  Use: 'G[n]'.
 |
 |      Parameters
 |      ----------
 |      n : node
 |         A node in the graph.
 |
 |      Returns
 |      -------
```

```
 |      adj_dict : dictionary
 |         The adjacency dictionary for nodes connected to n.
 |
 |      Notes
 |      -----
 |      G[n] is the same as G.adj[n] and similar to G.neighbors(n)
 |      (which is an iterator over G.adj[n])
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G[0]
 |      AtlasView({1: {}})
 |
 |  __iter__(self)
 |      Iterate over the nodes. Use: 'for n in G'.
 |
 |      Returns
 |      -------
 |      niter : iterator
 |          An iterator over all nodes in the graph.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> [n for n in G]
 |      [0, 1, 2, 3]
 |      >>> list(G)
 |      [0, 1, 2, 3]
 |
 |  __len__(self)
 |      Returns the number of nodes in the graph. Use: 'len(G)'.
 |
 |      Returns
 |      -------
 |      nnodes : int
 |          The number of nodes in the graph.
 |
 |      See Also
 |      --------
 |      number_of_nodes: identical method
 |      order: identical method
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> len(G)
 |      4
```

```
 |
 |  add_weighted_edges_from(self, ebunch_to_add, weight='weight', **attr)
 |      Add weighted edges in `ebunch_to_add` with specified weight attr
 |
 |      Parameters
 |      ----------
 |      ebunch_to_add : container of edges
 |          Each edge given in the list or container will be added
 |          to the graph. The edges must be given as 3-tuples (u, v, w)
 |          where w is a number.
 |      weight : string, optional (default= 'weight')
 |          The attribute name for the edge weights to be added.
 |      attr : keyword arguments, optional (default= no attributes)
 |          Edge attributes to add/update for all edges.
 |
 |      See Also
 |      --------
 |      add_edge : add a single edge
 |      add_edges_from : add multiple edges
 |
 |      Notes
 |      -----
 |      Adding the same edge twice for Graph/DiGraph simply updates
 |      the edge data. For MultiGraph/MultiDiGraph, duplicate edges
 |      are stored.
 |
 |      Examples
 |      --------
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
 |
 |  adjacency(self)
 |      Returns an iterator over (node, adjacency dict) tuples for all nodes.
 |
 |      For directed graphs, only outgoing neighbors/adjacencies are included.
 |
 |      Returns
 |      -------
 |      adj_iter : iterator
 |         An iterator over (node, adjacency dictionary) for all nodes in
 |         the graph.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
 |      [(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
 |
```

```
 |  copy(self, as_view=False)
 |      Returns a copy of the graph.
 |
 |      The copy method by default returns an independent shallow copy
 |      of the graph and attributes. That is, if an attribute is a
 |      container, that container is shared by the original an the copy.
 |      Use Python's `copy.deepcopy` for new containers.
 |
 |      If `as_view` is True then a view is returned instead of a copy.
 |
 |      Notes
 |      -----
 |      All copies reproduce the graph structure, but data attributes
 |      may be handled in different ways. There are four types of copies
 |      of a graph that people might want.
 |
 |      Deepcopy -- A "deepcopy" copies the graph structure as well as
 |      all data attributes and any objects they might contain.
 |      The entire graph object is new so that changes in the copy
 |      do not affect the original object. (see Python's copy.deepcopy)
 |
 |      Data Reference (Shallow) -- For a shallow copy the graph structure
 |      is copied but the edge, node and graph attribute dicts are
 |      references to those in the original graph. This saves
 |      time and memory but could cause confusion if you change an attribute
 |      in one graph and it changes the attribute in the other.
 |      NetworkX does not provide this level of shallow copy.
 |
 |      Independent Shallow -- This copy creates new independent attribute
 |      dicts and then does a shallow copy of the attributes. That is, any
 |      attributes that are containers are shared between the new graph
 |      and the original. This is exactly what `dict.copy()` provides.
 |      You can obtain this style copy using:
 |
 |          >>> G = nx.path_graph(5)
 |          >>> H = G.copy()
 |          >>> H = G.copy(as_view=False)
 |          >>> H = nx.Graph(G)
 |          >>> H = G.__class__(G)
 |
 |      Fresh Data -- For fresh data, the graph structure is copied while
 |      new empty data attribute dicts are created. The resulting graph
 |      is independent of the original and it has no edge, node or graph
 |      attributes. Fresh copies are not enabled. Instead use:
 |
 |          >>> H = G.__class__()
 |          >>> H.add_nodes_from(G)
 |          >>> H.add_edges_from(G.edges)
```

```
 |
 |      View -- Inspired by dict-views, graph-views act like read-only
 |      versions of the original graph, providing a copy of the original
 |      structure without requiring any memory for copying the information.
 |
 |      See the Python copy module for more information on shallow
 |      and deep copies, https://docs.python.org/3/library/copy.html.
 |
 |      Parameters
 |      ----------
 |      as_view : bool, optional (default=False)
 |          If True, the returned graph-view provides a read-only view
 |          of the original graph without actually copying any data.
 |
 |      Returns
 |      -------
 |      G : Graph
 |          A copy of the graph.
 |
 |      See Also
 |      --------
 |      to_directed: return a directed copy of the graph.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> H = G.copy()
 |
 |  edge_subgraph(self, edges)
 |      Returns the subgraph induced by the specified edges.
 |
 |      The induced subgraph contains each edge in `edges` and each
 |      node incident to any one of those edges.
 |
 |      Parameters
 |      ----------
 |      edges : iterable
 |          An iterable of edges in this graph.
 |
 |      Returns
 |      -------
 |      G : Graph
 |          An edge-induced subgraph of this graph with the same edge
 |          attributes.
 |
 |      Notes
 |      -----
 |      The graph, edge, and node attributes in the returned subgraph
```

```
|      view are references to the corresponding attributes in the original
|      graph. The view is read-only.
|
|      To create a full graph version of the subgraph with its own copy
|      of the edge or node attributes, use::
|
|          G.edge_subgraph(edges).copy()
|
|      Examples
|      --------
|      >>> G = nx.path_graph(5)
|      >>> H = G.edge_subgraph([(0, 1), (3, 4)])
|      >>> list(H.nodes)
|      [0, 1, 3, 4]
|      >>> list(H.edges)
|      [(0, 1), (3, 4)]
|
|  get_edge_data(self, u, v, default=None)
|      Returns the attribute dictionary associated with edge (u, v).
|
|      This is identical to `G[u][v]` except the default is returned
|      instead of an exception if the edge doesn't exist.
|
|      Parameters
|      ----------
|      u, v : nodes
|      default:  any Python object (default=None)
|          Value to return if the edge (u, v) is not found.
|
|      Returns
|      -------
|      edge_dict : dictionary
|          The edge attribute dictionary.
|
|      Examples
|      --------
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> G[0][1]
|      {}
|
|      Warning: Assigning to `G[u][v]` is not permitted.
|      But it is safe to assign attributes `G[u][v]['foo']`
|
|      >>> G[0][1]["weight"] = 7
|      >>> G[0][1]["weight"]
|      7
|      >>> G[1][0]["weight"]
|      7
```

```
|
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> G.get_edge_data(0, 1)  # default edge data is {}
|      {}
|      >>> e = (0, 1)
|      >>> G.get_edge_data(*e)  # tuple form
|      {}
|      >>> G.get_edge_data("a", "b", default=0)  # edge not in graph, return 0
|      0
|
|  has_edge(self, u, v)
|      Returns True if the edge (u, v) is in the graph.
|
|      This is the same as `v in G[u]` without KeyError exceptions.
|
|      Parameters
|      ----------
|      u, v : nodes
|          Nodes can be, for example, strings or numbers.
|          Nodes must be hashable (and not None) Python objects.
|
|      Returns
|      -------
|      edge_ind : bool
|          True if edge is in the graph, False otherwise.
|
|      Examples
|      --------
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> G.has_edge(0, 1)  # using two nodes
|      True
|      >>> e = (0, 1)
|      >>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
|      True
|      >>> e = (0, 1, {"weight": 7})
|      >>> G.has_edge(*e[:2])  # e is a 3-tuple (u, v, data_dictionary)
|      True
|
|      The following syntax are equivalent:
|
|      >>> G.has_edge(0, 1)
|      True
|      >>> 1 in G[0]  # though this gives KeyError if 0 not in G
|      True
|
|  has_node(self, n)
|      Returns True if the graph contains the node n.
|
```

```
 |      Identical to `n in G`
 |
 |      Parameters
 |      ----------
 |      n : node
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.has_node(0)
 |      True
 |
 |      It is more readable and simpler to use
 |
 |      >>> 0 in G
 |      True
 |
 |  nbunch_iter(self, nbunch=None)
 |      Returns an iterator over nodes contained in nbunch that are
 |      also in the graph.
 |
 |      The nodes in nbunch are checked for membership in the graph
 |      and if not are silently ignored.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges incident to these nodes.
 |
 |      Returns
 |      -------
 |      niter : iterator
 |          An iterator over nodes in nbunch that are also in the graph.
 |          If nbunch is None, iterate over all nodes in the graph.
 |
 |      Raises
 |      ------
 |      NetworkXError
 |          If nbunch is not a node or sequence of nodes.
 |          If a node in nbunch is not hashable.
 |
 |      See Also
 |      --------
 |      Graph.__iter__
 |
 |      Notes
 |      -----
 |      When nbunch is an iterator, the returned iterator yields values
```

```
 |      directly from nbunch, becoming exhausted when nbunch is exhausted.
 |
 |      To test whether nbunch is a single node, one can use
 |      "if nbunch in self:", even after processing with this routine.
 |
 |      If nbunch is not a node or a (possibly empty) sequence/iterator
 |      or None, a :exc:`NetworkXError` is raised.  Also, if any object in
 |      nbunch is not hashable, a :exc:`NetworkXError` is raised.
 |
 |  nodes = <functools.cached_property object>
 |      A NodeView of the Graph as G.nodes or G.nodes().
 |
 |      Can be used as `G.nodes` for data lookup and for set-like operations.
 |      Can also be used as `G.nodes(data='color', default=None)` to return a
 |      NodeDataView which reports specific node data but no set operations.
 |      It presents a dict-like interface as well with `G.nodes.items()`
 |      iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']`
 |      providing the value of the `foo` attribute for node `3`. In addition,
 |      a view `G.nodes.data('foo')` provides a dict-like interface to the
 |      `foo` attribute of each node. `G.nodes.data('foo', default=1)`
 |      provides a default for nodes that do not have attribute `foo`.
 |
 |      Parameters
 |      ----------
 |      data : string or bool, optional (default=False)
 |          The node attribute returned in 2-tuple (n, ddict[data]).
 |          If True, return entire node attribute dict as (n, ddict).
 |          If False, return just the nodes n.
 |
 |      default : value, optional (default=None)
 |          Value used for nodes that don't have the requested attribute.
 |          Only relevant if data is not True or False.
 |
 |      Returns
 |      -------
 |      NodeView
 |          Allows set-like operations over the nodes as well as node
 |          attribute dict lookup and calling to get a NodeDataView.
 |          A NodeDataView iterates over `(n, data)` and has no set operations.
 |          A NodeView iterates over `n` and includes set operations.
 |
 |          When called, if data is False, an iterator over nodes.
 |          Otherwise an iterator of 2-tuples (node, attribute value)
 |          where the attribute is specified in `data`.
 |          If data is True then the attribute becomes the
 |          entire data dictionary.
 |
 |      Notes
```

```
|       -----
|       If your node data is not needed, it is simpler and equivalent
|       to use the expression ``for n in G``, or ``list(G)``.
|
|       Examples
|       --------
|       There are two simple ways of getting a list of all nodes in the graph:
|
|       >>> G = nx.path_graph(3)
|       >>> list(G.nodes)
|       [0, 1, 2]
|       >>> list(G)
|       [0, 1, 2]
|
|       To get the node data along with the nodes:
|
|       >>> G.add_node(1, time="5pm")
|       >>> G.nodes[0]["foo"] = "bar"
|       >>> list(G.nodes(data=True))
|       [(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
|       >>> list(G.nodes.data())
|       [(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
|
|       >>> list(G.nodes(data="foo"))
|       [(0, 'bar'), (1, None), (2, None)]
|       >>> list(G.nodes.data("foo"))
|       [(0, 'bar'), (1, None), (2, None)]
|
|       >>> list(G.nodes(data="time"))
|       [(0, None), (1, '5pm'), (2, None)]
|       >>> list(G.nodes.data("time"))
|       [(0, None), (1, '5pm'), (2, None)]
|
|       >>> list(G.nodes(data="time", default="Not Available"))
|       [(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
|       >>> list(G.nodes.data("time", default="Not Available"))
|       [(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
|
|       If some of your nodes have an attribute and the rest are assumed
|       to have a default attribute value you can create a dictionary
|       from node/attribute pairs using the `default` keyword argument
|       to guarantee the value is never None::
|
|           >>> G = nx.Graph()
|           >>> G.add_node(0)
|           >>> G.add_node(1, weight=2)
|           >>> G.add_node(2, weight=3)
|           >>> dict(G.nodes(data="weight", default=1))
```

```
|         {0: 1, 1: 2, 2: 3}
|
|  number_of_edges(self, u=None, v=None)
|      Returns the number of edges between two nodes.
|
|      Parameters
|      ----------
|      u, v : nodes, optional (default=all edges)
|          If u and v are specified, return the number of edges between
|          u and v. Otherwise return the total number of all edges.
|
|      Returns
|      -------
|      nedges : int
|          The number of edges in the graph.  If nodes `u` and `v` are
|          specified return the number of edges between those nodes. If
|          the graph is directed, this only returns the number of edges
|          from `u` to `v`.
|
|      See Also
|      --------
|      size
|
|      Examples
|      --------
|      For undirected graphs, this method counts the total number of
|      edges in the graph:
|
|      >>> G = nx.path_graph(4)
|      >>> G.number_of_edges()
|      3
|
|      If you specify two nodes, this counts the total number of edges
|      joining the two nodes:
|
|      >>> G.number_of_edges(0, 1)
|      1
|
|      For directed graphs, this method can count the total number of
|      directed edges from `u` to `v`:
|
|      >>> G = nx.DiGraph()
|      >>> G.add_edge(0, 1)
|      >>> G.add_edge(1, 0)
|      >>> G.number_of_edges(0, 1)
|      1
|
|  number_of_nodes(self)
```

```
 |      Returns the number of nodes in the graph.
 |
 |      Returns
 |      -------
 |      nnodes : int
 |          The number of nodes in the graph.
 |
 |      See Also
 |      --------
 |      order: identical method
 |      __len__: identical method
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.number_of_nodes()
 |      3
 |
 |  order(self)
 |      Returns the number of nodes in the graph.
 |
 |      Returns
 |      -------
 |      nnodes : int
 |          The number of nodes in the graph.
 |
 |      See Also
 |      --------
 |      number_of_nodes: identical method
 |      __len__: identical method
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.order()
 |      3
 |
 |  size(self, weight=None)
 |      Returns the number of edges or total of all edge weights.
 |
 |      Parameters
 |      ----------
 |      weight : string or None, optional (default=None)
 |          The edge attribute that holds the numerical value used
 |          as a weight. If None, then each edge has weight 1.
 |
 |      Returns
 |      -------
```

```
 |      size : numeric
 |          The number of edges or
 |          (if weight keyword is provided) the total weight sum.
 |
 |          If weight is None, returns an int. Otherwise a float
 |          (or more general numeric if the weights are more general).
 |
 |      See Also
 |      --------
 |      number_of_edges
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.size()
 |      3
 |
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_edge("a", "b", weight=2)
 |      >>> G.add_edge("b", "c", weight=4)
 |      >>> G.size()
 |      2
 |      >>> G.size(weight="weight")
 |      6.0
 |
 |  subgraph(self, nodes)
 |      Returns a SubGraph view of the subgraph induced on `nodes`.
 |
 |      The induced subgraph of the graph contains the nodes in `nodes`
 |      and the edges between those nodes.
 |
 |      Parameters
 |      ----------
 |      nodes : list, iterable
 |          A container of nodes which will be iterated through once.
 |
 |      Returns
 |      -------
 |      G : SubGraph View
 |          A subgraph view of the graph. The graph structure cannot be
 |          changed but node/edge attributes can and are shared with the
 |          original graph.
 |
 |      Notes
 |      -----
 |      The graph, edge and node attributes are shared with the original graph.
 |      Changes to the graph structure is ruled out by the view, but changes
 |      to attributes are reflected in the original graph.
```

```
|
|      To create a subgraph with its own copy of the edge/node attributes use:
|      G.subgraph(nodes).copy()
|
|      For an inplace reduction of a graph to a subgraph you can remove nodes:
|      G.remove_nodes_from([n for n in G if n not in set(nodes)])
|
|      Subgraph views are sometimes NOT what you want. In most cases where
|      you want to do more than simply look at the induced edges, it makes
|      more sense to just create the subgraph as its own graph with code like:
|
|      ::
|
|          # Create a subgraph SG based on a (possibly multigraph) G
|          SG = G.__class__()
|          SG.add_nodes_from((n, G.nodes[n]) for n in largest_wcc)
|          if SG.is_multigraph():
|              SG.add_edges_from((n, nbr, key, d)
|                  for n, nbrs in G.adj.items() if n in largest_wcc
|                  for nbr, keydict in nbrs.items() if nbr in largest_wcc
|                  for key, d in keydict.items())
|          else:
|              SG.add_edges_from((n, nbr, d)
|                  for n, nbrs in G.adj.items() if n in largest_wcc
|                  for nbr, d in nbrs.items() if nbr in largest_wcc)
|          SG.graph.update(G.graph)
|
|      Examples
|      --------
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> H = G.subgraph([0, 1, 2])
|      >>> list(H.edges)
|      [(0, 1), (1, 2)]
|
|  to_directed(self, as_view=False)
|      Returns a directed representation of the graph.
|
|      Returns
|      -------
|      G : DiGraph
|          A directed graph with the same name, same nodes, and with
|          each edge (u, v, data) replaced by two directed edges
|          (u, v, data) and (v, u, data).
|
|      Notes
|      -----
|      This returns a "deepcopy" of the edge, node, and
|      graph attributes which attempts to completely copy
```

```
|      all of the data and references.
|
|      This is in contrast to the similar D=DiGraph(G) which returns a
|      shallow copy of the data.
|
|      See the Python copy module for more information on shallow
|      and deep copies, https://docs.python.org/3/library/copy.html.
|
|      Warning: If you have subclassed Graph to use dict-like objects
|      in the data structure, those changes do not transfer to the
|      DiGraph created by this method.
|
|      Examples
|      --------
|      >>> G = nx.Graph()  # or MultiGraph, etc
|      >>> G.add_edge(0, 1)
|      >>> H = G.to_directed()
|      >>> list(H.edges)
|      [(0, 1), (1, 0)]
|
|      If already directed, return a (deep) copy
|
|      >>> G = nx.DiGraph()  # or MultiDiGraph, etc
|      >>> G.add_edge(0, 1)
|      >>> H = G.to_directed()
|      >>> list(H.edges)
|      [(0, 1)]
|
|  to_directed_class(self)
|      Returns the class to use for empty directed copies.
|
|      If you subclass the base classes, use this to designate
|      what directed class to use for `to_directed()` copies.
|
|  to_undirected_class(self)
|      Returns the class to use for empty undirected copies.
|
|      If you subclass the base classes, use this to designate
|      what directed class to use for `to_directed()` copies.
|
|  update(self, edges=None, nodes=None)
|      Update the graph using nodes/edges/graphs as input.
|
|      Like dict.update, this method takes a graph as input, adding the
|      graph's nodes and edges to this graph. It can also take two inputs:
|      edges and nodes. Finally it can take either edges or nodes.
|      To specify only nodes the keyword `nodes` must be used.
|
```

```
|      The collections of edges and nodes are treated similarly to
|      the add_edges_from/add_nodes_from methods. When iterated, they
|      should yield 2-tuples (u, v) or 3-tuples (u, v, datadict).
|
|      Parameters
|      ----------
|      edges : Graph object, collection of edges, or None
|          The first parameter can be a graph or some edges. If it has
|          attributes `nodes` and `edges`, then it is taken to be a
|          Graph-like object and those attributes are used as collections
|          of nodes and edges to be added to the graph.
|          If the first parameter does not have those attributes, it is
|          treated as a collection of edges and added to the graph.
|          If the first argument is None, no edges are added.
|      nodes : collection of nodes, or None
|          The second parameter is treated as a collection of nodes
|          to be added to the graph unless it is None.
|          If `edges is None` and `nodes is None` an exception is raised.
|          If the first parameter is a Graph, then `nodes` is ignored.
|
|      Examples
|      --------
|      >>> G = nx.path_graph(5)
|      >>> G.update(nx.complete_graph(range(4, 10)))
|      >>> from itertools import combinations
|      >>> edges = (
|      …     (u, v, {"power": u * v})
|      …     for u, v in combinations(range(10, 20), 2)
|      …     if u * v < 225
|      … )
|      >>> nodes = [1000]  # for singleton, use a container
|      >>> G.update(edges, nodes)
|
|      Notes
|      -----
|      It you want to update the graph using an adjacency structure
|      it is straightforward to obtain the edges/nodes from adjacency.
|      The following examples provide common cases, your adjacency may
|      be slightly different and require tweaks of these examples::
|
|      >>> # dict-of-set/list/tuple
|      >>> adj = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}}
|      >>> e = [(u, v) for u, nbrs in adj.items() for v in nbrs]
|      >>> G.update(edges=e, nodes=adj)
|
|      >>> DG = nx.DiGraph()
|      >>> # dict-of-dict-of-attribute
|      >>> adj = {1: {2: 1.3, 3: 0.7}, 2: {1: 1.4}, 3: {1: 0.7}}
```

```
 |      >>> e = [
 |      …     (u, v, {"weight": d})
 |      …     for u, nbrs in adj.items()
 |      …     for v, d in nbrs.items()
 |      … ]
 |      >>> DG.update(edges=e, nodes=adj)
 |
 |      >>> # dict-of-dict-of-dict
 |      >>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
 |      >>> e = [
 |      …     (u, v, {"weight": d})
 |      …     for u, nbrs in adj.items()
 |      …     for v, d in nbrs.items()
 |      … ]
 |      >>> DG.update(edges=e, nodes=adj)
 |
 |      >>> # predecessor adjacency (dict-of-set)
 |      >>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
 |      >>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
 |
 |      >>> # MultiGraph dict-of-dict-of-dict-of-attribute
 |      >>> MDG = nx.MultiDiGraph()
 |      >>> adj = {
 |      …     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
 |      …     3: {2: {0: {"weight": 0.7}}},
 |      … }
 |      >>> e = [
 |      …     (u, v, ekey, d)
 |      …     for u, nbrs in adj.items()
 |      …     for v, keydict in nbrs.items()
 |      …     for ekey, d in keydict.items()
 |      … ]
 |      >>> MDG.update(edges=e)
 |
 |      See Also
 |      --------
 |      add_edges_from: add multiple edges to a graph
 |      add_nodes_from: add multiple nodes to a graph
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from networkx.classes.graph.Graph:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
```

```
 |  name
 |      String identifier of the graph.
 |
 |      This graph attribute appears in the attribute dict G.graph
 |      keyed by the string `"name"`. as well as an attribute (technically
 |      a property) `G.name`. This is entirely user controlled.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from networkx.classes.graph.Graph:
 |
 |  adjlist_inner_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  adjlist_outer_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  edge_attr_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  graph_attr_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
```

```
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  node_attr_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  node_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
```

[282]:

```
cust = CustomerNode(
    pk='id',
    prefix='cust',
    fpath='dat/cust.csv',
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
)

order = OrderNode(
    pk='id',
    prefix='order',
    fpath='dat/orders.csv',
    fmt='csv',
```

```
    compute_layer=ComputeLayerEnum.pandas,
    date_key='ts'
)
```

2023-10-25 18:36:07 [warning] no `date_key` set for <GraphReduceNode:
fpath=dat/cust.csv fmt=csv>

[283]:
```
gr = GraphReduce(
    name='odsc_first_graph',
    parent_node=cust,
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
    compute_period_val=365,
    compute_period_unit=PeriodUnit.day,
    label_period_val=30,
    label_period_unit=PeriodUnit.day,
    cut_date=datetime.datetime(2023, 6, 12),
    has_labels=True
)
```

[284]:
```
# show how attribute push down works
# gr.hydrate_graph_attrs
```

[285]:
```
gr.add_node(cust)
gr.add_node(order)
```

[286]:
```
gr.hydrate_graph_attrs()
```

2023-10-25 18:36:12 [info   ] hydrating attributes for CustomerNode
2023-10-25 18:36:12 [info   ] hydrating attributes for OrderNode

[287]:
```
order.compute_period_val
```

[287]: 365

[288]:
```
order.cut_date
```

[288]: datetime.datetime(2023, 6, 12, 0, 0)

[289]:
```
# add an edge
```

[290]:
```
help(gr.add_entity_edge)
```

Help on method add_entity_edge in module graphreduce.graph_reduce:

add_entity_edge(parent_node: graphreduce.node.GraphReduceNode, relation_node:
graphreduce.node.GraphReduceNode, parent_key: str, relation_key: str,
relation_type: str = 'parent_child', reduce: bool = True) method of

```
graphreduce.graph_reduce.GraphReduce instance
    Add an entity relation
```

[291]:
```python
gr.add_entity_edge(
    parent_node=cust,
    relation_node=order,
    parent_key='id',
    relation_key='customer_id',
    reduce=True
)
```

[292]:
```python
gr.do_transformations()
```

```
2023-10-25 18:36:15 [info     ] hydrating graph attributes
2023-10-25 18:36:15 [info     ] hydrating attributes for CustomerNode
2023-10-25 18:36:15 [info     ] hydrating attributes for OrderNode
2023-10-25 18:36:15 [info     ] hydrating graph data
2023-10-25 18:36:15 [info     ] checking for prefix uniqueness
2023-10-25 18:36:15 [info     ] running filters, normalize, and annotations for
<GraphReduceNode: fpath=dat/cust.csv fmt=csv>
2023-10-25 18:36:15 [info     ] running filters, normalize, and annotations for
<GraphReduceNode: fpath=dat/orders.csv fmt=csv>
2023-10-25 18:36:15 [info     ] depth-first traversal through the graph from
source: <GraphReduceNode: fpath=dat/cust.csv fmt=csv>
2023-10-25 18:36:15 [info     ] reducing relation <GraphReduceNode:
fpath=dat/orders.csv fmt=csv>
2023-10-25 18:36:15 [info     ] joining <GraphReduceNode: fpath=dat/orders.csv
fmt=csv> to <GraphReduceNode: fpath=dat/cust.csv fmt=csv>
2023-10-25 18:36:15 [info     ] computed labels for <GraphReduceNode:
fpath=dat/orders.csv fmt=csv>
```

[293]:
```python
gr.parent_node.df
```

[293]:
```
   cust_id cust_name  cust_name_length  order_customer_id  order_id_count  \
0        1       wes                 3                  1               2
1        2      john                 4                  2               2

   order_amount_sum  order_customer_id_dupe  order_id_had_order
0              21.5                       1                   1
1             250.0                       2                   1
```

[ ]:
```python

```

7 Constructing a graph without reducing relations.

[268]:
```
cust = CustomerNode(
    pk='id',
    prefix='cust',
    fpath='dat/cust.csv',
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
)

order = OrderNode(
    pk='id',
    prefix='order',
    fpath='dat/orders.csv',
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
    date_key='ts'
)
gr = GraphReduce(
    name='odsc_first_graph',
    parent_node=cust,
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
    compute_period_val=365,
    compute_period_unit=PeriodUnit.day,
    label_period_val=30,
    label_period_unit=PeriodUnit.day,
    cut_date=datetime.datetime(2023, 6, 12)
)

gr.add_node(cust)
gr.add_node(order)

gr.add_entity_edge(
    parent_node=cust,
    relation_node=order,
    parent_key='id',
    relation_key='customer_id',
    reduce=False
)
```

```
2023-10-25 18:35:17 [warning  ] no `date_key` set for <GraphReduceNode:
fpath=dat/cust.csv fmt=csv>
```

[269]:
```
gr.do_transformations()
```

```
2023-10-25 18:35:24 [info     ] hydrating graph attributes
2023-10-25 18:35:24 [info     ] hydrating attributes for CustomerNode
```

```
2023-10-25 18:35:24 [info     ] hydrating attributes for OrderNode
2023-10-25 18:35:24 [info     ] hydrating graph data
2023-10-25 18:35:24 [info     ] checking for prefix uniqueness
2023-10-25 18:35:24 [info     ] running filters, normalize, and annotations for
<GraphReduceNode: fpath=dat/cust.csv fmt=csv>
2023-10-25 18:35:24 [info     ] running filters, normalize, and annotations for
<GraphReduceNode: fpath=dat/orders.csv fmt=csv>
2023-10-25 18:35:24 [info     ] depth-first traversal through the graph from
source: <GraphReduceNode: fpath=dat/cust.csv fmt=csv>
2023-10-25 18:35:24 [info     ] doing nothing with relation node
<GraphReduceNode: fpath=dat/orders.csv fmt=csv>
2023-10-25 18:35:24 [info     ] joining <GraphReduceNode: fpath=dat/orders.csv
fmt=csv> to <GraphReduceNode: fpath=dat/cust.csv fmt=csv>
```

[270]:
```
gr.parent_node.df
```

[270]:
```
   cust_id cust_name  cust_name_length  order_id  order_customer_id  \
0        1       wes                 3         1                  1
1        1       wes                 3         2                  1
2        1       wes                 3         5                  1
3        1       wes                 3         7                  1
4        2      john                 4         3                  2
5        2      john                 4         4                  2
6        2      john                 4         6                  2

     order_ts  order_amount
0  2023-05-12          10.0
1  2023-06-01          11.5
2  2023-07-01         325.0
3  2023-07-14       12000.0
4  2023-01-01         100.0
5  2022-08-05         150.0
6  2023-07-02          23.0
```

8 Constructing a graph and automating feature generation.

[294]:
```
help(GraphReduce)
```

```
Help on class GraphReduce in module graphreduce.graph_reduce:

class GraphReduce(networkx.classes.digraph.DiGraph)
 |  GraphReduce(name: str = 'graph_reduce', parent_node:
Optional[graphreduce.node.GraphReduceNode] = None, fmt: str = 'parquet',
compute_layer: graphreduce.enum.ComputeLayerEnum = None, cut_date:
datetime.datetime = datetime.datetime(2023, 10, 25, 16, 41, 54, 934704),
compute_period_val: Union[int, float] = 365, compute_period_unit:
graphreduce.enum.PeriodUnit = <PeriodUnit.day: 'day'>, has_labels: bool = False,
```

```
label_period_val: Union[int, float, NoneType] = None, label_period_unit:
Optional[graphreduce.enum.PeriodUnit] = None, spark_sqlctx:
pyspark.sql.context.SQLContext = None, feature_function: Optional[str] = None,
dynamic_propagation: bool = False, type_func_map: Dict[str, List[str]] =
{'int64': ['min', 'max', 'sum'], 'str': ['first'], 'object': ['first'],
'float64': ['min', 'max', 'sum'], 'bool': ['first'], 'datetime64': ['first']},
storage_client: Optional[graphreduce.storage.StorageClient] = None, *args,
**kwargs)
 |
 |  Method resolution order:
 |      GraphReduce
 |      networkx.classes.digraph.DiGraph
 |      networkx.classes.graph.Graph
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, name: str = 'graph_reduce', parent_node:
Optional[graphreduce.node.GraphReduceNode] = None, fmt: str = 'parquet',
compute_layer: graphreduce.enum.ComputeLayerEnum = None, cut_date:
datetime.datetime = datetime.datetime(2023, 10, 25, 16, 41, 54, 934704),
compute_period_val: Union[int, float] = 365, compute_period_unit:
graphreduce.enum.PeriodUnit = <PeriodUnit.day: 'day'>, has_labels: bool = False,
label_period_val: Union[int, float, NoneType] = None, label_period_unit:
Optional[graphreduce.enum.PeriodUnit] = None, spark_sqlctx:
pyspark.sql.context.SQLContext = None, feature_function: Optional[str] = None,
dynamic_propagation: bool = False, type_func_map: Dict[str, List[str]] =
{'int64': ['min', 'max', 'sum'], 'str': ['first'], 'object': ['first'],
'float64': ['min', 'max', 'sum'], 'bool': ['first'], 'datetime64': ['first']},
storage_client: Optional[graphreduce.storage.StorageClient] = None, *args,
**kwargs)
 |      Constructor for GraphReduce
 |
 |      Args:
 |          name : the name of the graph reduce
 |          parent_node : parent-most node in the graph, if doing reductions the
granularity to which to reduce the data
 |          fmt : the format of the dataset
 |          compute_layer : compute layer to use (e.g., spark)
 |          cut_date : the date to cut off history
 |          compute_period_val : the amount of time to consider during the
compute job
 |          compute_period_unit : the unit for the compute period value (e.g.,
day)
 |          has_labels : whether or not the compute job computes labels, when
True `prep_for_labels()` and `compute_labels` will be called
 |          label_period_val : amount of time to consider when computing labels
 |          label_period_unit : the unit for the label period value (e.g., day)
```

```
 |          spark_sqlctx : if compute layer is spark this must be passed
 |          feature_function : optional custom feature function
 |          dynamic_propagation : optional to dynamically propagate children
data upward, useful for very large compute graphs
 |          type_func_match : optional mapping from type to a list of functions
(e.g., {'int' : ['min', 'max', 'sum'], 'str' : ['first']})
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  __str__(self)
 |      Returns a short summary of the graph.
 |
 |      Returns
 |      -------
 |      info : string
 |          Graph information as provided by `nx.info`
 |
 |      Examples
 |      --------
 |      >>> G = nx.Graph(name="foo")
 |      >>> str(G)
 |      "Graph named 'foo' with 0 nodes and 0 edges"
 |
 |      >>> G = nx.path_graph(3)
 |      >>> str(G)
 |      'Graph with 3 nodes and 2 edges'
 |
 |  add_entity_edge(self, parent_node: graphreduce.node.GraphReduceNode,
relation_node: graphreduce.node.GraphReduceNode, parent_key: str, relation_key:
str, relation_type: str = 'parent_child', reduce: bool = True)
 |      Add an entity relation
 |
 |  assign_parent(self, parent_node: graphreduce.node.GraphReduceNode)
 |      Assign the parent-most node in the graph
 |
 |  depth_first_generator(self)
 |      Depth-first traversal over the edges
 |
 |  do_transformations(self)
 |      Perform all graph transformations
 |      1) hydrate graph
 |      2) check for duplicate prefixes
 |      3) filter data
 |      4) clip anomalies
 |      5) annotate data
 |      6) depth-first edge traversal to: aggregate / reduce features and labels
 |      6a) optional alternative feature_function mapping
```

```
 |      6b) join back to parent edge
 |      6c) post-join annotations if any
 |      7) repeat step 6 on all edges up the hierarchy
 |
 |  get_children(self, node: graphreduce.node.GraphReduceNode) ->
List[graphreduce.node.GraphReduceNode]
 |      Get the children of a given node
 |
 |  hydrate_graph_attrs(self, attrs=['cut_date', 'compute_period_val',
'compute_period_unit', 'has_labels', 'label_period_val', 'label_period_unit',
'compute_layer', 'feature_function', 'spark_sqlctx', '_storage_client'])
 |      Hydrate the nodes in the graph with parent
 |      attributes in `attrs`
 |
 |  hydrate_graph_data(self)
 |      Hydrate the nodes in the graph with their data
 |
 |  join(self, parent_node: graphreduce.node.GraphReduceNode, relation_node:
graphreduce.node.GraphReduceNode, relation_df=None)
 |      Join the child or peer nnode to the parent node
 |
 |      Optionally pass the `child_df` directly
 |
 |  plot_graph(self, fname: str = 'graph.html')
 |      Plot the graph
 |
 |      Args
 |          fname : file name to save the graph to - should be .html
 |          notebook : whether or not to render in notebook
 |
 |  prefix_uniqueness(self)
 |      Identify children with duplicate prefixes, if any
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  parent
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from networkx.classes.digraph.DiGraph:
 |
 |  add_edge(self, u_of_edge, v_of_edge, **attr)
 |      Add an edge between u and v.
 |
 |      The nodes u and v will be automatically added if they are
 |      not already in the graph.
 |
 |      Edge attributes can be specified with keywords or by directly
```

```
|      accessing the edge's attribute dictionary. See examples below.
|
|      Parameters
|      ----------
|      u_of_edge, v_of_edge : nodes
|          Nodes can be, for example, strings or numbers.
|          Nodes must be hashable (and not None) Python objects.
|      attr : keyword arguments, optional
|          Edge data (or labels or objects) can be assigned using
|          keyword arguments.
|
|      See Also
|      --------
|      add_edges_from : add a collection of edges
|
|      Notes
|      -----
|      Adding an edge that already exists updates the edge data.
|
|      Many NetworkX algorithms designed for weighted graphs use
|      an edge attribute (by default `weight`) to hold a numerical value.
|
|      Examples
|      --------
|      The following all add the edge e=(1, 2) to graph G:
|
|      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> e = (1, 2)
|      >>> G.add_edge(1, 2)  # explicit two-node form
|      >>> G.add_edge(*e)  # single edge as tuple of two nodes
|      >>> G.add_edges_from([(1, 2)])  # add edges from iterable container
|
|      Associate data to edges using keywords:
|
|      >>> G.add_edge(1, 2, weight=3)
|      >>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
|
|      For non-string attribute keys, use subscript notation.
|
|      >>> G.add_edge(1, 2)
|      >>> G[1][2].update({0: 5})
|      >>> G.edges[1, 2].update({0: 5})
|
|  add_edges_from(self, ebunch_to_add, **attr)
|      Add all the edges in ebunch_to_add.
|
|      Parameters
|      ----------
```

```
 |      ebunch_to_add : container of edges
 |          Each edge given in the container will be added to the
 |          graph. The edges must be given as 2-tuples (u, v) or
 |          3-tuples (u, v, d) where d is a dictionary containing edge data.
 |      attr : keyword arguments, optional
 |          Edge data (or labels or objects) can be assigned using
 |          keyword arguments.
 |
 |      See Also
 |      --------
 |      add_edge : add a single edge
 |      add_weighted_edges_from : convenient way to add weighted edges
 |
 |      Notes
 |      -----
 |      Adding the same edge twice has no effect but any edge data
 |      will be updated when each duplicate edge is added.
 |
 |      Edge attributes specified in an ebunch take precedence over
 |      attributes specified via keyword arguments.
 |
 |      Examples
 |      --------
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_edges_from([(0, 1), (1, 2)])  # using a list of edge tuples
 |      >>> e = zip(range(0, 3), range(1, 4))
 |      >>> G.add_edges_from(e)  # Add the path graph 0-1-2-3
 |
 |      Associate data to edges
 |
 |      >>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
 |      >>> G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
 |
 |  add_node(self, node_for_adding, **attr)
 |      Add a single node `node_for_adding` and update node attributes.
 |
 |      Parameters
 |      ----------
 |      node_for_adding : node
 |          A node can be any hashable Python object except None.
 |      attr : keyword arguments, optional
 |          Set or change node attributes using key=value.
 |
 |      See Also
 |      --------
 |      add_nodes_from
 |
 |      Examples
```

```
 |      --------
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_node(1)
 |      >>> G.add_node("Hello")
 |      >>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
 |      >>> G.add_node(K3)
 |      >>> G.number_of_nodes()
 |      3
 |
 |      Use keywords set/change node attributes:
 |
 |      >>> G.add_node(1, size=10)
 |      >>> G.add_node(3, weight=0.4, UTM=("13S", 382871, 3972649))
 |
 |      Notes
 |      -----
 |      A hashable object is one that can be used as a key in a Python
 |      dictionary. This includes strings, numbers, tuples of strings
 |      and numbers, etc.
 |
 |      On many platforms hashable items also include mutables such as
 |      NetworkX Graphs, though one should be careful that the hash
 |      doesn't change on mutables.
 |
 |  add_nodes_from(self, nodes_for_adding, **attr)
 |      Add multiple nodes.
 |
 |      Parameters
 |      ----------
 |      nodes_for_adding : iterable container
 |          A container of nodes (list, dict, set, etc.).
 |          OR
 |          A container of (node, attribute dict) tuples.
 |          Node attributes are updated using the attribute dict.
 |      attr : keyword arguments, optional (default= no attributes)
 |          Update attributes for all nodes in nodes.
 |          Node attributes specified in nodes as a tuple take
 |          precedence over attributes specified via keyword arguments.
 |
 |      See Also
 |      --------
 |      add_node
 |
 |      Examples
 |      --------
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_nodes_from("Hello")
 |      >>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
```

```
 |      >>> G.add_nodes_from(K3)
 |      >>> sorted(G.nodes(), key=str)
 |      [0, 1, 2, 'H', 'e', 'l', 'o']
 |
 |      Use keywords to update specific node attributes for every node.
 |
 |      >>> G.add_nodes_from([1, 2], size=10)
 |      >>> G.add_nodes_from([3, 4], weight=0.4)
 |
 |      Use (node, attrdict) tuples to update attributes for specific nodes.
 |
 |      >>> G.add_nodes_from([(1, dict(size=11)), (2, {"color": "blue"})])
 |      >>> G.nodes[1]["size"]
 |      11
 |      >>> H = nx.Graph()
 |      >>> H.add_nodes_from(G.nodes(data=True))
 |      >>> H.nodes[1]["size"]
 |      11
 |
 |  adj = <functools.cached_property object>
 |      Graph adjacency object holding the neighbors of each node.
 |
 |      This object is a read-only dict-like structure with node keys
 |      and neighbor-dict values.  The neighbor-dict is keyed by neighbor
 |      to the edge-data-dict.  So `G.adj[3][2]['color'] = 'blue'` sets
 |      the color of the edge `(3, 2)` to `"blue"`.
 |
 |      Iterating over G.adj behaves like a dict. Useful idioms include
 |      `for nbr, datadict in G.adj[n].items():`.
 |
 |      The neighbor information is also provided by subscripting the graph.
 |      So `for nbr, foovalue in G[node].data('foo', default=1):` works.
 |
 |      For directed graphs, `G.adj` holds outgoing (successor) info.
 |
 |  clear(self)
 |      Remove all nodes and edges from the graph.
 |
 |      This also removes the name, and all graph, node, and edge attributes.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.clear()
 |      >>> list(G.nodes)
 |      []
 |      >>> list(G.edges)
 |      []
```

```
 |
 |  clear_edges(self)
 |      Remove all edges from the graph without altering nodes.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.clear_edges()
 |      >>> list(G.nodes)
 |      [0, 1, 2, 3]
 |      >>> list(G.edges)
 |      []
 |
 |  degree = <functools.cached_property object>
 |      A DegreeView for the Graph as G.degree or G.degree().
 |
 |      The node degree is the number of edges adjacent to the node.
 |      The weighted node degree is the sum of the edge weights for
 |      edges incident to that node.
 |
 |      This object provides an iterator for (node, degree) as well as
 |      lookup for the degree for a single node.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges incident to these nodes.
 |
 |      weight : string or None, optional (default=None)
 |         The name of an edge attribute that holds the numerical value used
 |         as a weight.  If None, then each edge has weight 1.
 |         The degree is the sum of the edge weights adjacent to the node.
 |
 |      Returns
 |      -------
 |      DiDegreeView or int
 |          If multiple nodes are requested (the default), returns a
`DiDegreeView`
 |          mapping nodes to their degree.
 |          If a single node is requested, returns the degree of the node as an
integer.
 |
 |      See Also
 |      --------
 |      in_degree, out_degree
 |
 |      Examples
 |      --------
```

```
 |      >>> G = nx.DiGraph()  # or MultiDiGraph
 |      >>> nx.add_path(G, [0, 1, 2, 3])
 |      >>> G.degree(0)  # node 0 with degree 1
 |      1
 |      >>> list(G.degree([0, 1, 2]))
 |      [(0, 1), (1, 2), (2, 2)]
 |
 |  edges = <functools.cached_property object>
 |      An OutEdgeView of the DiGraph as G.edges or G.edges().
 |
 |      edges(self, nbunch=None, data=False, default=None)
 |
 |      The OutEdgeView provides set-like operations on the edge-tuples
 |      as well as edge attribute lookup. When called, it also provides
 |      an EdgeDataView object which allows control of access to edge
 |      attributes (but does not provide set-like operations).
 |      Hence, `G.edges[u, v]['color']` provides the value of the color
 |      attribute for edge `(u, v)` while
 |      `for (u, v, c) in G.edges.data('color', default='red'):`
 |      iterates through all the edges yielding the color attribute
 |      with default `'red'` if no color attribute exists.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges from these nodes.
 |      data : string or bool, optional (default=False)
 |          The edge attribute returned in 3-tuple (u, v, ddict[data]).
 |          If True, return edge attribute dict in 3-tuple (u, v, ddict).
 |          If False, return 2-tuple (u, v).
 |      default : value, optional (default=None)
 |          Value used for edges that don't have the requested attribute.
 |          Only relevant if data is not True or False.
 |
 |      Returns
 |      -------
 |      edges : OutEdgeView
 |          A view of edge attributes, usually it iterates over (u, v)
 |          or (u, v, d) tuples of edges, but can also be used for
 |          attribute lookup as `edges[u, v]['foo']`.
 |
 |      See Also
 |      --------
 |      in_edges, out_edges
 |
 |      Notes
 |      -----
 |      Nodes in nbunch that are not in the graph will be (quietly) ignored.
```

```
 |      For directed graphs this returns the out-edges.
 |
 |      Examples
 |      --------
 |      >>> G = nx.DiGraph()  # or MultiDiGraph, etc
 |      >>> nx.add_path(G, [0, 1, 2])
 |      >>> G.add_edge(2, 3, weight=5)
 |      >>> [e for e in G.edges]
 |      [(0, 1), (1, 2), (2, 3)]
 |      >>> G.edges.data()  # default data is {} (empty dict)
 |      OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
 |      >>> G.edges.data("weight", default=1)
 |      OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
 |      >>> G.edges([0, 2])  # only edges originating from these nodes
 |      OutEdgeDataView([(0, 1), (2, 3)])
 |      >>> G.edges(0)  # only edges from node 0
 |      OutEdgeDataView([(0, 1)])
 |
 |  has_predecessor(self, u, v)
 |      Returns True if node u has predecessor v.
 |
 |      This is true if graph has the edge u<-v.
 |
 |  has_successor(self, u, v)
 |      Returns True if node u has successor v.
 |
 |      This is true if graph has the edge u->v.
 |
 |  in_degree = <functools.cached_property object>
 |      An InDegreeView for (node, in_degree) or in_degree for single node.
 |
 |      The node in_degree is the number of edges pointing to the node.
 |      The weighted node degree is the sum of the edge weights for
 |      edges incident to that node.
 |
 |      This object provides an iteration over (node, in_degree) as well as
 |      lookup for the degree for a single node.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges incident to these nodes.
 |
 |      weight : string or None, optional (default=None)
 |         The name of an edge attribute that holds the numerical value used
 |         as a weight.  If None, then each edge has weight 1.
 |         The degree is the sum of the edge weights adjacent to the node.
 |
```

```
 |      Returns
 |      -------
 |      If a single node is requested
 |      deg : int
 |          In-degree of the node
 |
 |      OR if multiple nodes are requested
 |      nd_iter : iterator
 |          The iterator returns two-tuples of (node, in-degree).
 |
 |      See Also
 |      --------
 |      degree, out_degree
 |
 |      Examples
 |      --------
 |      >>> G = nx.DiGraph()
 |      >>> nx.add_path(G, [0, 1, 2, 3])
 |      >>> G.in_degree(0)  # node 0 with degree 0
 |      0
 |      >>> list(G.in_degree([0, 1, 2]))
 |      [(0, 0), (1, 1), (2, 1)]
 |
 |  in_edges = <functools.cached_property object>
 |      An InEdgeView of the Graph as G.in_edges or G.in_edges().
 |
 |      in_edges(self, nbunch=None, data=False, default=None):
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges incident to these nodes.
 |      data : string or bool, optional (default=False)
 |          The edge attribute returned in 3-tuple (u, v, ddict[data]).
 |          If True, return edge attribute dict in 3-tuple (u, v, ddict).
 |          If False, return 2-tuple (u, v).
 |      default : value, optional (default=None)
 |          Value used for edges that don't have the requested attribute.
 |          Only relevant if data is not True or False.
 |
 |      Returns
 |      -------
 |      in_edges : InEdgeView
 |          A view of edge attributes, usually it iterates over (u, v)
 |          or (u, v, d) tuples of edges, but can also be used for
 |          attribute lookup as `edges[u, v]['foo']`.
 |
 |      See Also
```

```
|      --------
|      edges
|
|  is_directed(self)
|      Returns True if graph is directed, False otherwise.
|
|  is_multigraph(self)
|      Returns True if graph is a multigraph, False otherwise.
|
|  neighbors = successors(self, n)
|
|  out_degree = <functools.cached_property object>
|      An OutDegreeView for (node, out_degree)
|
|      The node out_degree is the number of edges pointing out of the node.
|      The weighted node degree is the sum of the edge weights for
|      edges incident to that node.
|
|      This object provides an iterator over (node, out_degree) as well as
|      lookup for the degree for a single node.
|
|      Parameters
|      ----------
|      nbunch : single node, container, or all nodes (default= all nodes)
|          The view will only report edges incident to these nodes.
|
|      weight : string or None, optional (default=None)
|         The name of an edge attribute that holds the numerical value used
|         as a weight.  If None, then each edge has weight 1.
|         The degree is the sum of the edge weights adjacent to the node.
|
|      Returns
|      -------
|      If a single node is requested
|      deg : int
|          Out-degree of the node
|
|      OR if multiple nodes are requested
|      nd_iter : iterator
|          The iterator returns two-tuples of (node, out-degree).
|
|      See Also
|      --------
|      degree, in_degree
|
|      Examples
|      --------
|      >>> G = nx.DiGraph()
```

```
 |      >>> nx.add_path(G, [0, 1, 2, 3])
 |      >>> G.out_degree(0)  # node 0 with degree 1
 |      1
 |      >>> list(G.out_degree([0, 1, 2]))
 |      [(0, 1), (1, 1), (2, 1)]
 |
 |  out_edges = <functools.cached_property object>
 |      An OutEdgeView of the DiGraph as G.edges or G.edges().
 |
 |      edges(self, nbunch=None, data=False, default=None)
 |
 |      The OutEdgeView provides set-like operations on the edge-tuples
 |      as well as edge attribute lookup. When called, it also provides
 |      an EdgeDataView object which allows control of access to edge
 |      attributes (but does not provide set-like operations).
 |      Hence, `G.edges[u, v]['color']` provides the value of the color
 |      attribute for edge `(u, v)` while
 |      `for (u, v, c) in G.edges.data('color', default='red'):`
 |      iterates through all the edges yielding the color attribute
 |      with default `'red'` if no color attribute exists.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges from these nodes.
 |      data : string or bool, optional (default=False)
 |          The edge attribute returned in 3-tuple (u, v, ddict[data]).
 |          If True, return edge attribute dict in 3-tuple (u, v, ddict).
 |          If False, return 2-tuple (u, v).
 |      default : value, optional (default=None)
 |          Value used for edges that don't have the requested attribute.
 |          Only relevant if data is not True or False.
 |
 |      Returns
 |      -------
 |      edges : OutEdgeView
 |          A view of edge attributes, usually it iterates over (u, v)
 |          or (u, v, d) tuples of edges, but can also be used for
 |          attribute lookup as `edges[u, v]['foo']`.
 |
 |      See Also
 |      --------
 |      in_edges, out_edges
 |
 |      Notes
 |      -----
 |      Nodes in nbunch that are not in the graph will be (quietly) ignored.
 |      For directed graphs this returns the out-edges.
```

```
|
|      Examples
|      --------
|      >>> G = nx.DiGraph()  # or MultiDiGraph, etc
|      >>> nx.add_path(G, [0, 1, 2])
|      >>> G.add_edge(2, 3, weight=5)
|      >>> [e for e in G.edges]
|      [(0, 1), (1, 2), (2, 3)]
|      >>> G.edges.data()  # default data is {} (empty dict)
|      OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
|      >>> G.edges.data("weight", default=1)
|      OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
|      >>> G.edges([0, 2])  # only edges originating from these nodes
|      OutEdgeDataView([(0, 1), (2, 3)])
|      >>> G.edges(0)  # only edges from node 0
|      OutEdgeDataView([(0, 1)])
|
|  pred = <functools.cached_property object>
|      Graph adjacency object holding the predecessors of each node.
|
|      This object is a read-only dict-like structure with node keys
|      and neighbor-dict values.  The neighbor-dict is keyed by neighbor
|      to the edge-data-dict.  So `G.pred[2][3]['color'] = 'blue'` sets
|      the color of the edge `(3, 2)` to `"blue"`.
|
|      Iterating over G.pred behaves like a dict. Useful idioms include
|      `for nbr, datadict in G.pred[n].items():`.  A data-view not provided
|      by dicts also exists: `for nbr, foovalue in G.pred[node].data('foo'):`
|      A default can be set via a `default` argument to the `data` method.
|
|  predecessors(self, n)
|      Returns an iterator over predecessor nodes of n.
|
|      A predecessor of n is a node m such that there exists a directed
|      edge from m to n.
|
|      Parameters
|      ----------
|      n : node
|         A node in the graph
|
|      Raises
|      ------
|      NetworkXError
|         If n is not in the graph.
|
|      See Also
|      --------
```

```
|      successors
|
|  remove_edge(self, u, v)
|      Remove the edge between u and v.
|
|      Parameters
|      ----------
|      u, v : nodes
|          Remove the edge between nodes u and v.
|
|      Raises
|      ------
|      NetworkXError
|          If there is not an edge between u and v.
|
|      See Also
|      --------
|      remove_edges_from : remove a collection of edges
|
|      Examples
|      --------
|      >>> G = nx.Graph()  # or DiGraph, etc
|      >>> nx.add_path(G, [0, 1, 2, 3])
|      >>> G.remove_edge(0, 1)
|      >>> e = (1, 2)
|      >>> G.remove_edge(*e)  # unpacks e from an edge tuple
|      >>> e = (2, 3, {"weight": 7})  # an edge with attribute data
|      >>> G.remove_edge(*e[:2])  # select first part of edge tuple
|
|  remove_edges_from(self, ebunch)
|      Remove all edges specified in ebunch.
|
|      Parameters
|      ----------
|      ebunch: list or container of edge tuples
|          Each edge given in the list or container will be removed
|          from the graph. The edges can be:
|
|              - 2-tuples (u, v) edge between u and v.
|              - 3-tuples (u, v, k) where k is ignored.
|
|      See Also
|      --------
|      remove_edge : remove a single edge
|
|      Notes
|      -----
|      Will fail silently if an edge in ebunch is not in the graph.
```

```
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> ebunch = [(1, 2), (2, 3)]
 |      >>> G.remove_edges_from(ebunch)
 |
 |  remove_node(self, n)
 |      Remove node n.
 |
 |      Removes the node n and all adjacent edges.
 |      Attempting to remove a non-existent node will raise an exception.
 |
 |      Parameters
 |      ----------
 |      n : node
 |         A node in the graph
 |
 |      Raises
 |      ------
 |      NetworkXError
 |         If n is not in the graph.
 |
 |      See Also
 |      --------
 |      remove_nodes_from
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> list(G.edges)
 |      [(0, 1), (1, 2)]
 |      >>> G.remove_node(1)
 |      >>> list(G.edges)
 |      []
 |
 |  remove_nodes_from(self, nodes)
 |      Remove multiple nodes.
 |
 |      Parameters
 |      ----------
 |      nodes : iterable container
 |          A container of nodes (list, dict, set, etc.).  If a node
 |          in the container is not in the graph it is silently ignored.
 |
 |      See Also
 |      --------
 |      remove_node
```

```
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> e = list(G.nodes)
 |      >>> e
 |      [0, 1, 2]
 |      >>> G.remove_nodes_from(e)
 |      >>> list(G.nodes)
 |      []
 |
 |  reverse(self, copy=True)
 |      Returns the reverse of the graph.
 |
 |      The reverse is a graph with the same nodes and edges
 |      but with the directions of the edges reversed.
 |
 |      Parameters
 |      ----------
 |      copy : bool optional (default=True)
 |          If True, return a new DiGraph holding the reversed edges.
 |          If False, the reverse graph is created using a view of
 |          the original graph.
 |
 |  succ = <functools.cached_property object>
 |      Graph adjacency object holding the successors of each node.
 |
 |      This object is a read-only dict-like structure with node keys
 |      and neighbor-dict values.  The neighbor-dict is keyed by neighbor
 |      to the edge-data-dict.  So `G.succ[3][2]['color'] = 'blue'` sets
 |      the color of the edge `(3, 2)` to `"blue"`.
 |
 |      Iterating over G.succ behaves like a dict. Useful idioms include
 |      `for nbr, datadict in G.succ[n].items():`.  A data-view not provided
 |      by dicts also exists: `for nbr, foovalue in G.succ[node].data('foo'):`
 |      and a default can be set via a `default` argument to the `data` method.
 |
 |      The neighbor information is also provided by subscripting the graph.
 |      So `for nbr, foovalue in G[node].data('foo', default=1):` works.
 |
 |      For directed graphs, `G.adj` is identical to `G.succ`.
 |
 |  successors(self, n)
 |      Returns an iterator over successor nodes of n.
 |
 |      A successor of n is a node m such that there exists a directed
 |      edge from n to m.
 |
```

```
 |      Parameters
 |      ----------
 |      n : node
 |         A node in the graph
 |
 |      Raises
 |      ------
 |      NetworkXError
 |         If n is not in the graph.
 |
 |      See Also
 |      --------
 |      predecessors
 |
 |      Notes
 |      -----
 |      neighbors() and successors() are the same.
 |
 |  to_undirected(self, reciprocal=False, as_view=False)
 |      Returns an undirected representation of the digraph.
 |
 |      Parameters
 |      ----------
 |      reciprocal : bool (optional)
 |        If True only keep edges that appear in both directions
 |        in the original digraph.
 |      as_view : bool (optional, default=False)
 |        If True return an undirected view of the original directed graph.
 |
 |      Returns
 |      -------
 |      G : Graph
 |          An undirected graph with the same name and nodes and
 |          with edge (u, v, data) if either (u, v, data) or (v, u, data)
 |          is in the digraph.  If both edges exist in digraph and
 |          their edge data is different, only one edge is created
 |          with an arbitrary choice of which edge data to use.
 |          You must check and correct for this manually if desired.
 |
 |      See Also
 |      --------
 |      Graph, copy, add_edge, add_edges_from
 |
 |      Notes
 |      -----
 |      If edges in both directions (u, v) and (v, u) exist in the
 |      graph, attributes for the new undirected edge will be a combination of
 |      the attributes of the directed edges.  The edge data is updated
```

```
 |      in the (arbitrary) order that the edges are encountered.  For
 |      more customized control of the edge attributes use add_edge().
 |
 |      This returns a "deepcopy" of the edge, node, and
 |      graph attributes which attempts to completely copy
 |      all of the data and references.
 |
 |      This is in contrast to the similar G=DiGraph(D) which returns a
 |      shallow copy of the data.
 |
 |      See the Python copy module for more information on shallow
 |      and deep copies, https://docs.python.org/3/library/copy.html.
 |
 |      Warning: If you have subclassed DiGraph to use dict-like objects
 |      in the data structure, those changes do not transfer to the
 |      Graph created by this method.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(2)  # or MultiGraph, etc
 |      >>> H = G.to_directed()
 |      >>> list(H.edges)
 |      [(0, 1), (1, 0)]
 |      >>> G2 = H.to_undirected()
 |      >>> list(G2.edges)
 |      [(0, 1)]
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from networkx.classes.graph.Graph:
 |
 |  __contains__(self, n)
 |      Returns True if n is a node, False otherwise. Use: 'n in G'.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> 1 in G
 |      True
 |
 |  __getitem__(self, n)
 |      Returns a dict of neighbors of node n.  Use: 'G[n]'.
 |
 |      Parameters
 |      ----------
 |      n : node
 |         A node in the graph.
 |
 |      Returns
```

```
|      -------
|      adj_dict : dictionary
|         The adjacency dictionary for nodes connected to n.
|
|      Notes
|      -----
|      G[n] is the same as G.adj[n] and similar to G.neighbors(n)
|      (which is an iterator over G.adj[n])
|
|      Examples
|      --------
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> G[0]
|      AtlasView({1: {}})
|
|  __iter__(self)
|      Iterate over the nodes. Use: 'for n in G'.
|
|      Returns
|      -------
|      niter : iterator
|          An iterator over all nodes in the graph.
|
|      Examples
|      --------
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> [n for n in G]
|      [0, 1, 2, 3]
|      >>> list(G)
|      [0, 1, 2, 3]
|
|  __len__(self)
|      Returns the number of nodes in the graph. Use: 'len(G)'.
|
|      Returns
|      -------
|      nnodes : int
|          The number of nodes in the graph.
|
|      See Also
|      --------
|      number_of_nodes: identical method
|      order: identical method
|
|      Examples
|      --------
|      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> len(G)
```

```
 |      4
 |
 |  add_weighted_edges_from(self, ebunch_to_add, weight='weight', **attr)
 |      Add weighted edges in `ebunch_to_add` with specified weight attr
 |
 |      Parameters
 |      ----------
 |      ebunch_to_add : container of edges
 |          Each edge given in the list or container will be added
 |          to the graph. The edges must be given as 3-tuples (u, v, w)
 |          where w is a number.
 |      weight : string, optional (default= 'weight')
 |          The attribute name for the edge weights to be added.
 |      attr : keyword arguments, optional (default= no attributes)
 |          Edge attributes to add/update for all edges.
 |
 |      See Also
 |      --------
 |      add_edge : add a single edge
 |      add_edges_from : add multiple edges
 |
 |      Notes
 |      -----
 |      Adding the same edge twice for Graph/DiGraph simply updates
 |      the edge data. For MultiGraph/MultiDiGraph, duplicate edges
 |      are stored.
 |
 |      Examples
 |      --------
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
 |
 |  adjacency(self)
 |      Returns an iterator over (node, adjacency dict) tuples for all nodes.
 |
 |      For directed graphs, only outgoing neighbors/adjacencies are included.
 |
 |      Returns
 |      -------
 |      adj_iter : iterator
 |         An iterator over (node, adjacency dictionary) for all nodes in
 |         the graph.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
 |      [(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

```
 |
 |  copy(self, as_view=False)
 |      Returns a copy of the graph.
 |
 |      The copy method by default returns an independent shallow copy
 |      of the graph and attributes. That is, if an attribute is a
 |      container, that container is shared by the original an the copy.
 |      Use Python's `copy.deepcopy` for new containers.
 |
 |      If `as_view` is True then a view is returned instead of a copy.
 |
 |      Notes
 |      -----
 |      All copies reproduce the graph structure, but data attributes
 |      may be handled in different ways. There are four types of copies
 |      of a graph that people might want.
 |
 |      Deepcopy -- A "deepcopy" copies the graph structure as well as
 |      all data attributes and any objects they might contain.
 |      The entire graph object is new so that changes in the copy
 |      do not affect the original object. (see Python's copy.deepcopy)
 |
 |      Data Reference (Shallow) -- For a shallow copy the graph structure
 |      is copied but the edge, node and graph attribute dicts are
 |      references to those in the original graph. This saves
 |      time and memory but could cause confusion if you change an attribute
 |      in one graph and it changes the attribute in the other.
 |      NetworkX does not provide this level of shallow copy.
 |
 |      Independent Shallow -- This copy creates new independent attribute
 |      dicts and then does a shallow copy of the attributes. That is, any
 |      attributes that are containers are shared between the new graph
 |      and the original. This is exactly what `dict.copy()` provides.
 |      You can obtain this style copy using:
 |
 |          >>> G = nx.path_graph(5)
 |          >>> H = G.copy()
 |          >>> H = G.copy(as_view=False)
 |          >>> H = nx.Graph(G)
 |          >>> H = G.__class__(G)
 |
 |      Fresh Data -- For fresh data, the graph structure is copied while
 |      new empty data attribute dicts are created. The resulting graph
 |      is independent of the original and it has no edge, node or graph
 |      attributes. Fresh copies are not enabled. Instead use:
 |
 |          >>> H = G.__class__()
 |          >>> H.add_nodes_from(G)
```

```
 |          >>> H.add_edges_from(G.edges)
 |
 |      View -- Inspired by dict-views, graph-views act like read-only
 |      versions of the original graph, providing a copy of the original
 |      structure without requiring any memory for copying the information.
 |
 |      See the Python copy module for more information on shallow
 |      and deep copies, https://docs.python.org/3/library/copy.html.
 |
 |      Parameters
 |      ----------
 |      as_view : bool, optional (default=False)
 |          If True, the returned graph-view provides a read-only view
 |          of the original graph without actually copying any data.
 |
 |      Returns
 |      -------
 |      G : Graph
 |          A copy of the graph.
 |
 |      See Also
 |      --------
 |      to_directed: return a directed copy of the graph.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> H = G.copy()
 |
 |  edge_subgraph(self, edges)
 |      Returns the subgraph induced by the specified edges.
 |
 |      The induced subgraph contains each edge in `edges` and each
 |      node incident to any one of those edges.
 |
 |      Parameters
 |      ----------
 |      edges : iterable
 |          An iterable of edges in this graph.
 |
 |      Returns
 |      -------
 |      G : Graph
 |          An edge-induced subgraph of this graph with the same edge
 |          attributes.
 |
 |      Notes
 |      -----
```

```
 |      The graph, edge, and node attributes in the returned subgraph
 |      view are references to the corresponding attributes in the original
 |      graph. The view is read-only.
 |
 |      To create a full graph version of the subgraph with its own copy
 |      of the edge or node attributes, use::
 |
 |          G.edge_subgraph(edges).copy()
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(5)
 |      >>> H = G.edge_subgraph([(0, 1), (3, 4)])
 |      >>> list(H.nodes)
 |      [0, 1, 3, 4]
 |      >>> list(H.edges)
 |      [(0, 1), (3, 4)]
 |
 |  get_edge_data(self, u, v, default=None)
 |      Returns the attribute dictionary associated with edge (u, v).
 |
 |      This is identical to `G[u][v]` except the default is returned
 |      instead of an exception if the edge doesn't exist.
 |
 |      Parameters
 |      ----------
 |      u, v : nodes
 |      default:  any Python object (default=None)
 |          Value to return if the edge (u, v) is not found.
 |
 |      Returns
 |      -------
 |      edge_dict : dictionary
 |          The edge attribute dictionary.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G[0][1]
 |      {}
 |
 |      Warning: Assigning to `G[u][v]` is not permitted.
 |      But it is safe to assign attributes `G[u][v]['foo']`
 |
 |      >>> G[0][1]["weight"] = 7
 |      >>> G[0][1]["weight"]
 |      7
 |      >>> G[1][0]["weight"]
```

```
 |      7
 |
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.get_edge_data(0, 1)  # default edge data is {}
 |      {}
 |      >>> e = (0, 1)
 |      >>> G.get_edge_data(*e)  # tuple form
 |      {}
 |      >>> G.get_edge_data("a", "b", default=0)  # edge not in graph, return 0
 |      0
 |
 |  has_edge(self, u, v)
 |      Returns True if the edge (u, v) is in the graph.
 |
 |      This is the same as `v in G[u]` without KeyError exceptions.
 |
 |      Parameters
 |      ----------
 |      u, v : nodes
 |          Nodes can be, for example, strings or numbers.
 |          Nodes must be hashable (and not None) Python objects.
 |
 |      Returns
 |      -------
 |      edge_ind : bool
 |          True if edge is in the graph, False otherwise.
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.has_edge(0, 1)  # using two nodes
 |      True
 |      >>> e = (0, 1)
 |      >>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
 |      True
 |      >>> e = (0, 1, {"weight": 7})
 |      >>> G.has_edge(*e[:2])  # e is a 3-tuple (u, v, data_dictionary)
 |      True
 |
 |      The following syntax are equivalent:
 |
 |      >>> G.has_edge(0, 1)
 |      True
 |      >>> 1 in G[0]  # though this gives KeyError if 0 not in G
 |      True
 |
 |  has_node(self, n)
 |      Returns True if the graph contains the node n.
```

```
 |
 |      Identical to `n in G`
 |
 |      Parameters
 |      ----------
 |      n : node
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.has_node(0)
 |      True
 |
 |      It is more readable and simpler to use
 |
 |      >>> 0 in G
 |      True
 |
 |  nbunch_iter(self, nbunch=None)
 |      Returns an iterator over nodes contained in nbunch that are
 |      also in the graph.
 |
 |      The nodes in nbunch are checked for membership in the graph
 |      and if not are silently ignored.
 |
 |      Parameters
 |      ----------
 |      nbunch : single node, container, or all nodes (default= all nodes)
 |          The view will only report edges incident to these nodes.
 |
 |      Returns
 |      -------
 |      niter : iterator
 |          An iterator over nodes in nbunch that are also in the graph.
 |          If nbunch is None, iterate over all nodes in the graph.
 |
 |      Raises
 |      ------
 |      NetworkXError
 |          If nbunch is not a node or sequence of nodes.
 |          If a node in nbunch is not hashable.
 |
 |      See Also
 |      --------
 |      Graph.__iter__
 |
 |      Notes
 |      -----
```

```
 |      When nbunch is an iterator, the returned iterator yields values
 |      directly from nbunch, becoming exhausted when nbunch is exhausted.
 |
 |      To test whether nbunch is a single node, one can use
 |      "if nbunch in self:", even after processing with this routine.
 |
 |      If nbunch is not a node or a (possibly empty) sequence/iterator
 |      or None, a :exc:`NetworkXError` is raised.  Also, if any object in
 |      nbunch is not hashable, a :exc:`NetworkXError` is raised.
 |
 |  nodes = <functools.cached_property object>
 |      A NodeView of the Graph as G.nodes or G.nodes().
 |
 |      Can be used as `G.nodes` for data lookup and for set-like operations.
 |      Can also be used as `G.nodes(data='color', default=None)` to return a
 |      NodeDataView which reports specific node data but no set operations.
 |      It presents a dict-like interface as well with `G.nodes.items()`
 |      iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']`
 |      providing the value of the `foo` attribute for node `3`. In addition,
 |      a view `G.nodes.data('foo')` provides a dict-like interface to the
 |      `foo` attribute of each node. `G.nodes.data('foo', default=1)`
 |      provides a default for nodes that do not have attribute `foo`.
 |
 |      Parameters
 |      ----------
 |      data : string or bool, optional (default=False)
 |          The node attribute returned in 2-tuple (n, ddict[data]).
 |          If True, return entire node attribute dict as (n, ddict).
 |          If False, return just the nodes n.
 |
 |      default : value, optional (default=None)
 |          Value used for nodes that don't have the requested attribute.
 |          Only relevant if data is not True or False.
 |
 |      Returns
 |      -------
 |      NodeView
 |          Allows set-like operations over the nodes as well as node
 |          attribute dict lookup and calling to get a NodeDataView.
 |          A NodeDataView iterates over `(n, data)` and has no set operations.
 |          A NodeView iterates over `n` and includes set operations.
 |
 |          When called, if data is False, an iterator over nodes.
 |          Otherwise an iterator of 2-tuples (node, attribute value)
 |          where the attribute is specified in `data`.
 |          If data is True then the attribute becomes the
 |          entire data dictionary.
 |
```

```
|      Notes
|      -----
|      If your node data is not needed, it is simpler and equivalent
|      to use the expression ``for n in G``, or ``list(G)``.
|
|      Examples
|      --------
|      There are two simple ways of getting a list of all nodes in the graph:
|
|      >>> G = nx.path_graph(3)
|      >>> list(G.nodes)
|      [0, 1, 2]
|      >>> list(G)
|      [0, 1, 2]
|
|      To get the node data along with the nodes:
|
|      >>> G.add_node(1, time="5pm")
|      >>> G.nodes[0]["foo"] = "bar"
|      >>> list(G.nodes(data=True))
|      [(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
|      >>> list(G.nodes.data())
|      [(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
|
|      >>> list(G.nodes(data="foo"))
|      [(0, 'bar'), (1, None), (2, None)]
|      >>> list(G.nodes.data("foo"))
|      [(0, 'bar'), (1, None), (2, None)]
|
|      >>> list(G.nodes(data="time"))
|      [(0, None), (1, '5pm'), (2, None)]
|      >>> list(G.nodes.data("time"))
|      [(0, None), (1, '5pm'), (2, None)]
|
|      >>> list(G.nodes(data="time", default="Not Available"))
|      [(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
|      >>> list(G.nodes.data("time", default="Not Available"))
|      [(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
|
|      If some of your nodes have an attribute and the rest are assumed
|      to have a default attribute value you can create a dictionary
|      from node/attribute pairs using the `default` keyword argument
|      to guarantee the value is never None::
|
|          >>> G = nx.Graph()
|          >>> G.add_node(0)
|          >>> G.add_node(1, weight=2)
|          >>> G.add_node(2, weight=3)
```

```
 |          >>> dict(G.nodes(data="weight", default=1))
 |          {0: 1, 1: 2, 2: 3}
 |
 |  number_of_edges(self, u=None, v=None)
 |      Returns the number of edges between two nodes.
 |
 |      Parameters
 |      ----------
 |      u, v : nodes, optional (default=all edges)
 |          If u and v are specified, return the number of edges between
 |          u and v. Otherwise return the total number of all edges.
 |
 |      Returns
 |      -------
 |      nedges : int
 |          The number of edges in the graph.  If nodes `u` and `v` are
 |          specified return the number of edges between those nodes. If
 |          the graph is directed, this only returns the number of edges
 |          from `u` to `v`.
 |
 |      See Also
 |      --------
 |      size
 |
 |      Examples
 |      --------
 |      For undirected graphs, this method counts the total number of
 |      edges in the graph:
 |
 |      >>> G = nx.path_graph(4)
 |      >>> G.number_of_edges()
 |      3
 |
 |      If you specify two nodes, this counts the total number of edges
 |      joining the two nodes:
 |
 |      >>> G.number_of_edges(0, 1)
 |      1
 |
 |      For directed graphs, this method can count the total number of
 |      directed edges from `u` to `v`:
 |
 |      >>> G = nx.DiGraph()
 |      >>> G.add_edge(0, 1)
 |      >>> G.add_edge(1, 0)
 |      >>> G.number_of_edges(0, 1)
 |      1
 |
```

```
|  number_of_nodes(self)
|      Returns the number of nodes in the graph.
|
|      Returns
|      -------
|      nnodes : int
|          The number of nodes in the graph.
|
|      See Also
|      --------
|      order: identical method
|      __len__: identical method
|
|      Examples
|      --------
|      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> G.number_of_nodes()
|      3
|
|  order(self)
|      Returns the number of nodes in the graph.
|
|      Returns
|      -------
|      nnodes : int
|          The number of nodes in the graph.
|
|      See Also
|      --------
|      number_of_nodes: identical method
|      __len__: identical method
|
|      Examples
|      --------
|      >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
|      >>> G.order()
|      3
|
|  size(self, weight=None)
|      Returns the number of edges or total of all edge weights.
|
|      Parameters
|      ----------
|      weight : string or None, optional (default=None)
|          The edge attribute that holds the numerical value used
|          as a weight. If None, then each edge has weight 1.
|
|      Returns
```

```
 |      -------
 |      size : numeric
 |          The number of edges or
 |          (if weight keyword is provided) the total weight sum.
 |
 |          If weight is None, returns an int. Otherwise a float
 |          (or more general numeric if the weights are more general).
 |
 |      See Also
 |      --------
 |      number_of_edges
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.size()
 |      3
 |
 |      >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> G.add_edge("a", "b", weight=2)
 |      >>> G.add_edge("b", "c", weight=4)
 |      >>> G.size()
 |      2
 |      >>> G.size(weight="weight")
 |      6.0
 |
 |  subgraph(self, nodes)
 |      Returns a SubGraph view of the subgraph induced on `nodes`.
 |
 |      The induced subgraph of the graph contains the nodes in `nodes`
 |      and the edges between those nodes.
 |
 |      Parameters
 |      ----------
 |      nodes : list, iterable
 |          A container of nodes which will be iterated through once.
 |
 |      Returns
 |      -------
 |      G : SubGraph View
 |          A subgraph view of the graph. The graph structure cannot be
 |          changed but node/edge attributes can and are shared with the
 |          original graph.
 |
 |      Notes
 |      -----
 |      The graph, edge and node attributes are shared with the original graph.
 |      Changes to the graph structure is ruled out by the view, but changes
```

```
 |      to attributes are reflected in the original graph.
 |
 |      To create a subgraph with its own copy of the edge/node attributes use:
 |      G.subgraph(nodes).copy()
 |
 |      For an inplace reduction of a graph to a subgraph you can remove nodes:
 |      G.remove_nodes_from([n for n in G if n not in set(nodes)])
 |
 |      Subgraph views are sometimes NOT what you want. In most cases where
 |      you want to do more than simply look at the induced edges, it makes
 |      more sense to just create the subgraph as its own graph with code like:
 |
 |      ::
 |
 |          # Create a subgraph SG based on a (possibly multigraph) G
 |          SG = G.__class__()
 |          SG.add_nodes_from((n, G.nodes[n]) for n in largest_wcc)
 |          if SG.is_multigraph():
 |              SG.add_edges_from((n, nbr, key, d)
 |                  for n, nbrs in G.adj.items() if n in largest_wcc
 |                  for nbr, keydict in nbrs.items() if nbr in largest_wcc
 |                  for key, d in keydict.items())
 |          else:
 |              SG.add_edges_from((n, nbr, d)
 |                  for n, nbrs in G.adj.items() if n in largest_wcc
 |                  for nbr, d in nbrs.items() if nbr in largest_wcc)
 |          SG.graph.update(G.graph)
 |
 |      Examples
 |      --------
 |      >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
 |      >>> H = G.subgraph([0, 1, 2])
 |      >>> list(H.edges)
 |      [(0, 1), (1, 2)]
 |
 |  to_directed(self, as_view=False)
 |      Returns a directed representation of the graph.
 |
 |      Returns
 |      -------
 |      G : DiGraph
 |          A directed graph with the same name, same nodes, and with
 |          each edge (u, v, data) replaced by two directed edges
 |          (u, v, data) and (v, u, data).
 |
 |      Notes
 |      -----
 |      This returns a "deepcopy" of the edge, node, and
```

```
 |      graph attributes which attempts to completely copy
 |      all of the data and references.
 |
 |      This is in contrast to the similar D=DiGraph(G) which returns a
 |      shallow copy of the data.
 |
 |      See the Python copy module for more information on shallow
 |      and deep copies, https://docs.python.org/3/library/copy.html.
 |
 |      Warning: If you have subclassed Graph to use dict-like objects
 |      in the data structure, those changes do not transfer to the
 |      DiGraph created by this method.
 |
 |      Examples
 |      --------
 |      >>> G = nx.Graph()  # or MultiGraph, etc
 |      >>> G.add_edge(0, 1)
 |      >>> H = G.to_directed()
 |      >>> list(H.edges)
 |      [(0, 1), (1, 0)]
 |
 |      If already directed, return a (deep) copy
 |
 |      >>> G = nx.DiGraph()  # or MultiDiGraph, etc
 |      >>> G.add_edge(0, 1)
 |      >>> H = G.to_directed()
 |      >>> list(H.edges)
 |      [(0, 1)]
 |
 |  to_directed_class(self)
 |      Returns the class to use for empty directed copies.
 |
 |      If you subclass the base classes, use this to designate
 |      what directed class to use for `to_directed()` copies.
 |
 |  to_undirected_class(self)
 |      Returns the class to use for empty undirected copies.
 |
 |      If you subclass the base classes, use this to designate
 |      what directed class to use for `to_directed()` copies.
 |
 |  update(self, edges=None, nodes=None)
 |      Update the graph using nodes/edges/graphs as input.
 |
 |      Like dict.update, this method takes a graph as input, adding the
 |      graph's nodes and edges to this graph. It can also take two inputs:
 |      edges and nodes. Finally it can take either edges or nodes.
 |      To specify only nodes the keyword `nodes` must be used.
```

```
|
|      The collections of edges and nodes are treated similarly to
|      the add_edges_from/add_nodes_from methods. When iterated, they
|      should yield 2-tuples (u, v) or 3-tuples (u, v, datadict).
|
|      Parameters
|      ----------
|      edges : Graph object, collection of edges, or None
|          The first parameter can be a graph or some edges. If it has
|          attributes `nodes` and `edges`, then it is taken to be a
|          Graph-like object and those attributes are used as collections
|          of nodes and edges to be added to the graph.
|          If the first parameter does not have those attributes, it is
|          treated as a collection of edges and added to the graph.
|          If the first argument is None, no edges are added.
|      nodes : collection of nodes, or None
|          The second parameter is treated as a collection of nodes
|          to be added to the graph unless it is None.
|          If `edges is None` and `nodes is None` an exception is raised.
|          If the first parameter is a Graph, then `nodes` is ignored.
|
|      Examples
|      --------
|      >>> G = nx.path_graph(5)
|      >>> G.update(nx.complete_graph(range(4, 10)))
|      >>> from itertools import combinations
|      >>> edges = (
|      …     (u, v, {"power": u * v})
|      …     for u, v in combinations(range(10, 20), 2)
|      …     if u * v < 225
|      … )
|      >>> nodes = [1000]  # for singleton, use a container
|      >>> G.update(edges, nodes)
|
|      Notes
|      -----
|      It you want to update the graph using an adjacency structure
|      it is straightforward to obtain the edges/nodes from adjacency.
|      The following examples provide common cases, your adjacency may
|      be slightly different and require tweaks of these examples::
|
|      >>> # dict-of-set/list/tuple
|      >>> adj = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}}
|      >>> e = [(u, v) for u, nbrs in adj.items() for v in nbrs]
|      >>> G.update(edges=e, nodes=adj)
|
|      >>> DG = nx.DiGraph()
|      >>> # dict-of-dict-of-attribute
```

```
|      >>> adj = {1: {2: 1.3, 3: 0.7}, 2: {1: 1.4}, 3: {1: 0.7}}
|      >>> e = [
|      …     (u, v, {"weight": d})
|      …     for u, nbrs in adj.items()
|      …     for v, d in nbrs.items()
|      … ]
|      >>> DG.update(edges=e, nodes=adj)
|
|      >>> # dict-of-dict-of-dict
|      >>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
|      >>> e = [
|      …     (u, v, {"weight": d})
|      …     for u, nbrs in adj.items()
|      …     for v, d in nbrs.items()
|      … ]
|      >>> DG.update(edges=e, nodes=adj)
|
|      >>> # predecessor adjacency (dict-of-set)
|      >>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
|      >>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
|
|      >>> # MultiGraph dict-of-dict-of-dict-of-attribute
|      >>> MDG = nx.MultiDiGraph()
|      >>> adj = {
|      …     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
|      …     3: {2: {0: {"weight": 0.7}}},
|      … }
|      >>> e = [
|      …     (u, v, ekey, d)
|      …     for u, nbrs in adj.items()
|      …     for v, keydict in nbrs.items()
|      …     for ekey, d in keydict.items()
|      … ]
|      >>> MDG.update(edges=e)
|
|      See Also
|      --------
|      add_edges_from: add multiple edges to a graph
|      add_nodes_from: add multiple nodes to a graph
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from networkx.classes.graph.Graph:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

```
 |
 |  name
 |      String identifier of the graph.
 |
 |      This graph attribute appears in the attribute dict G.graph
 |      keyed by the string `"name"`. as well as an attribute (technically
 |      a property) `G.name`. This is entirely user controlled.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from networkx.classes.graph.Graph:
 |
 |  adjlist_inner_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  adjlist_outer_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  edge_attr_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  graph_attr_dict_factory = <class 'dict'>
```

```
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  node_attr_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |
 |  node_dict_factory = <class 'dict'>
 |      dict() -> new empty dictionary
 |      dict(mapping) -> new dictionary initialized from a mapping object's
 |          (key, value) pairs
 |      dict(iterable) -> new dictionary initialized as if via:
 |          d = {}
 |          for k, v in iterable:
 |              d[k] = v
 |      dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |          in the keyword argument list.  For example:  dict(one=1, two=2)
```

[301]:

```
cust = CustomerNode(
    pk='id',
    prefix='cust',
    fpath='dat/cust.csv',
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
)

order = OrderNode(
    pk='id',
    prefix='order',
    fpath='dat/orders.csv',
```

```
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
    date_key='ts'
)
gr = GraphReduce(
    name='odsc_first_graph',
    parent_node=cust,
    fmt='csv',
    compute_layer=ComputeLayerEnum.pandas,
    compute_period_val=365,
    compute_period_unit=PeriodUnit.day,
    label_period_val=30,
    label_period_unit=PeriodUnit.day,
    cut_date=datetime.datetime(2023, 6, 12),
    dynamic_propagation=True,
    has_labels=True
)

gr.add_node(cust)
gr.add_node(order)

gr.add_entity_edge(
    parent_node=cust,
    relation_node=order,
    parent_key='id',
    relation_key='customer_id',
    reduce=True
)
```

```
2023-10-25 18:37:39 [warning  ] no `date_key` set for <GraphReduceNode:
fpath=dat/cust.csv fmt=csv>
```

[302]:
```
gr.do_transformations()
```

```
2023-10-25 18:37:40 [info     ] hydrating graph attributes
2023-10-25 18:37:40 [info     ] hydrating attributes for CustomerNode
2023-10-25 18:37:40 [info     ] hydrating attributes for OrderNode
2023-10-25 18:37:40 [info     ] hydrating graph data
2023-10-25 18:37:40 [info     ] checking for prefix uniqueness
2023-10-25 18:37:40 [info     ] running filters, normalize, and annotations for
<GraphReduceNode: fpath=dat/cust.csv fmt=csv>
2023-10-25 18:37:40 [info     ] running filters, normalize, and annotations for
<GraphReduceNode: fpath=dat/orders.csv fmt=csv>
2023-10-25 18:37:40 [info     ] depth-first traversal through the graph from
source: <GraphReduceNode: fpath=dat/cust.csv fmt=csv>
2023-10-25 18:37:40 [info     ] reducing relation <GraphReduceNode:
fpath=dat/orders.csv fmt=csv>
2023-10-25 18:37:40 [info     ] doing dynamic propagation on node
```

```
<GraphReduceNode: fpath=dat/orders.csv fmt=csv>
2023-10-25 18:37:40 [info     ] joining <GraphReduceNode: fpath=dat/orders.csv
fmt=csv> to <GraphReduceNode: fpath=dat/cust.csv fmt=csv>
2023-10-25 18:37:40 [info     ] computed labels for <GraphReduceNode:
fpath=dat/orders.csv fmt=csv>
```

[303]:

```
gr.parent_node.df
```

[303]:

```
   cust_id cust_name  cust_name_length  order_customer_id  order_id_count  \
0        1       wes                 3                  1               2
1        2      john                 4                  2               2

   order_amount_sum  order_id_min  order_id_max  order_id_sum  \
0              21.5             1             2             3
1             250.0             3             4             7

   order_customer_id_min  order_customer_id_max  order_customer_id_sum  \
0                      1                      1                      2
1                      2                      2                      4

  order_ts_first  order_amount_min  order_amount_max  order_amount_sum_dupe  \
0     2023-05-12              10.0              11.5                   21.5
1     2023-01-01             100.0             150.0                  250.0

   order_customer_id_dupe  order_id_had_order
0                       1                   1
1                       2                   1
```

[ ]: