😎 Grafos con NetworkX

NetworkX es un paquete de Python para la creación, manipulación y estudio de estructuras dinámicas y funciones de redes complejas.

pip install networkx

Los grafos se utilizan para modelar situaciones, es una representación simplificada de la situación. Se aplican en Informática, Ciencias Sociales, Lingüística, Arquitectura, Comunicaciones, Física, Química, Ingeniería, etc.

Definición informal: conjunto de nodos unidos por aristas.

Definición formal: Un grafo es una terna G = (V , A , Φ), dónde:

V: Conjuntos de vértices, dónde $V \neq \emptyset$

A: Conjuntos de aristas.

Φ: Función de incidencia Φ: $A \rightarrow V^{(2)}$. Y $V^{(2)}$ es el conjunto formado por subconjuntos de 1 o 2 elementos de V, que son los extremos de la arista.

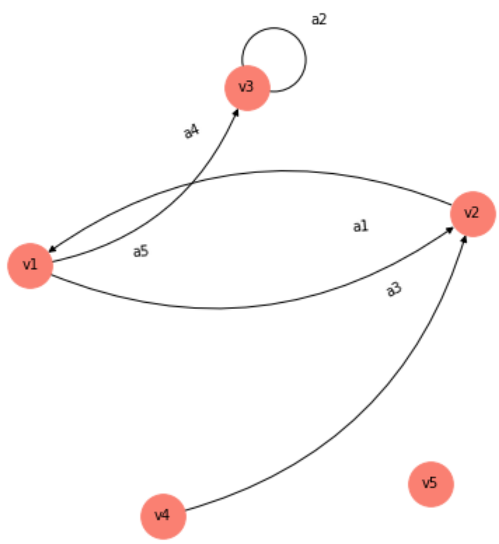
Ejemplo:

Sea el grafo G = (V , A , Φ), siendo los conjuntos:

- $V = \{v_1, v_2, v_3, v_4, v_5\}$
- $A = \{a_1, a_2, a_3, a_4, a_5\}$

Y la función de incidencia:

$\Phi(a_1) = \{v_1, v_2\}$,
$\Phi(a_2) = \{v_3\}$,
$\Phi(a_3) = \{v_4, v_2\}$,
$\Phi(a_4) = \{v_1, v_3\}$,
$\Phi(a_5) = \{v_1, v_2\}$



Vértice aislado: v_5 es aislado, $v_i \Leftrightarrow \forall\, v_k \in V$: Si $v_i \neq v_k$: v_i no es adyacente a v_k. Significa que es un vértice que no es adyacente a ningún otro.

Aristas paralelas: a_1 y a_5 son paralelas, a_1 es paralela a $a_k \Leftrightarrow \Phi(a_i) = \Phi(a_k)$. Significa que son aristas comprendidas entre los mismos vértices.

Bucles o lazos: a_2 es bucle o lazo, a_i es bucle o lazo $\Leftrightarrow |\Phi(a_i)| = 1$. Significa que es una arista con ambos extremos en el mismo vértice.

```
In [1]:
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

import warnings
warnings.filterwarnings('ignore')
```

Crear un grafo

Graph(): *Grafo vacío sin nodos ni aristas (un "grafo nulo") . G es el identificador para este ejemplo.*

```
In [2]:
G = nx.Graph()
```

La clase (Graph) implementa un grafo no dirigido. Si ejecutamos este ejemplo no obtendremos resultados porque el grafo está vacío, no dibujamos nodos ni enlaces y además no importamos el módulo para visualizar.

add_node(): *Con add.node agregamos un nodo y le pasamos como parámetro el valor.*

```
In [3]:
G = nx.Graph()
G.add_node('Hola Mundo!!')
```

draw(): *Con nx.draw dibujamos el nodo.*

```
In [4]:
G = nx.Graph()
G.add_node('Hola Mundo!!')
nx.draw(G)
```

Se pueden generar diferentes tipos de grafos, draw() es el común, también está el draw_circular, draw_espectral(), draw_shell(), etc.

draw_networkx():

```
In [5]:
G = nx.Graph()
G.add_node('Hola Mundo!!!')
nx.draw_networkx(G)
plt.axis('off')
plt.show()
```

Diferencia entre draw() y draw_networkx()

- **draw()** : dibuje el gráfico como una representación simple sin etiquetas de nodo o etiquetas de borde y utilizando el área de figura completa de Matplotlib sin etiquetas de eje de forma predeterminada.

- **draw_networkx()** : dibuje el gráfico con Matplotlib con opciones para posiciones de nodos, etiquetado, títulos y muchas otras características de dibujo.

[Documentación (https://networkx.org/documentation/stable/reference/drawing.html?highlight=draw!)](https://networkx.org/documentation/stable/reference/drawing.html?highlight=draw!)

Nodos

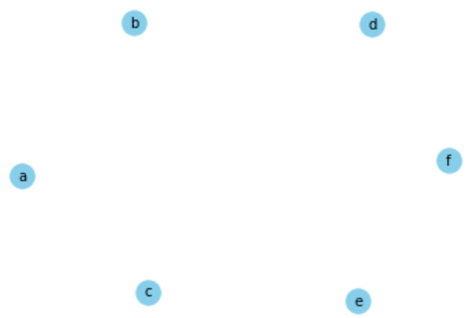
Agregar nodos

add_node() : *Agregamos nodos y visualizamos.*

In [6]:
```python
G = nx.Graph()
G.add_node('Nodo 1')
G.add_node('Nodo 2')
G.add_node('Nodo 3')
G.add_node('Nodo 4')
G.add_node('Nodo 5')
nx.draw_networkx(G, with_labels=True, node_size=300, font_size=10, node_color = 'skyblue')
plt.axis('off')
plt.show()
```

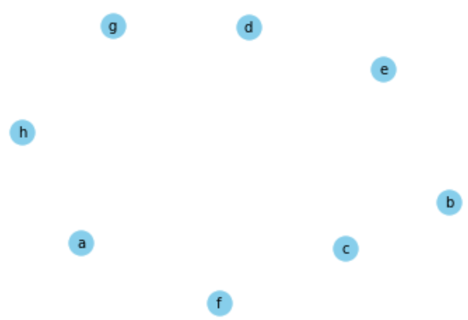
add_nodes_from() : *Agregamos nodos desde una lista y visualizamos.*

In [7]:
```python
L1 = ['a','b','c','d','e','f']
G = nx.Graph()
G.add_nodes_from(L1)
nx.draw_networkx(G, with_labels=True, node_size=300, font_size=10, node_color = 'skyblue')
plt.axis('off')
plt.show()
```



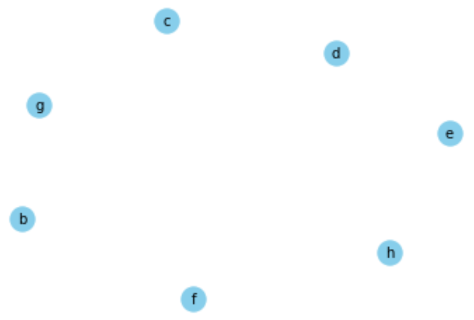
Eliminar nodos

In [8]:
```python
L1 = ['a','b','c','d','e','f', 'g', 'h']
G = nx.Graph()
G.add_nodes_from(L1)
nx.draw_networkx(G, with_labels=True, node_size=300, font_size=10, node_color = 'skyblue')
plt.axis('off')
plt.show()
```



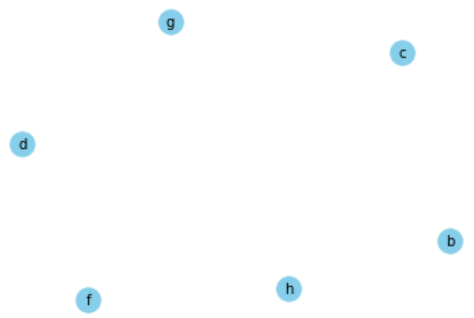
remove_node

In [9]:
```
G.remove_node('a')
nx.draw_networkx(G, with_labels=True, node_size=300, font_size=10, node_color = 'skyblue')
plt.axis('off')
plt.show()
```



remove_nodes_from: *Con remove_node o remove_nodes_from se eliminan nodos.*

In [10]:
```
G.remove_nodes_from('e')
nx.draw_networkx(G, with_labels=True, node_size=300, font_size=10, node_color = 'skyblue')
plt.axis('off')
plt.show()
```



Atributos de los nodos

nodes()

In [11]:
```
print("Nodos: ", G.nodes())
```

```
Nodos:  ['b', 'c', 'd', 'f', 'g', 'h']
```

number_of_nodes

In [12]:
```
print("Número de nodos: ", G.number_of_nodes())
```

```
Número de nodos:  6
```

data

Los atributos de los nodos se devuelven como un diccionarios. Y con:

- **list(G.nodes(data=True)) o**
- **list(G.nodes.data()) devuelve los atributos de los nodos como listas de tuplas compuestas del valor del nodo y un diccionario de sus atributos.**

In [13]:
```
G = nx.Graph()
G.add_node('Nodo 1',labels='Hola')
G.add_node('Nodo 2')
G.add_node('Nodo 3',labels='Cómo estás?')
list(G.nodes(data=True)), list(G.nodes.data())
```

Out[13]:
```
([('Nodo\xa01', {'labels': 'Hola'}),
  ('Nodo\xa02', {}),
  ('Nodo\xa03', {'labels': 'Cómo\xa0estás?'})],
 [('Nodo\xa01', {'labels': 'Hola'}),
  ('Nodo\xa02', {}),
  ('Nodo\xa03', {'labels': 'Cómo\xa0estás?'})])
```

with_labels:*Visualizamos las etiquetas del nodo.*

In [14]:
```
nx.draw(G, with_labels=True, width = 1000, node_color = 'skyblue')
```

node_size: *Modificamos el tamaño del nodo.*

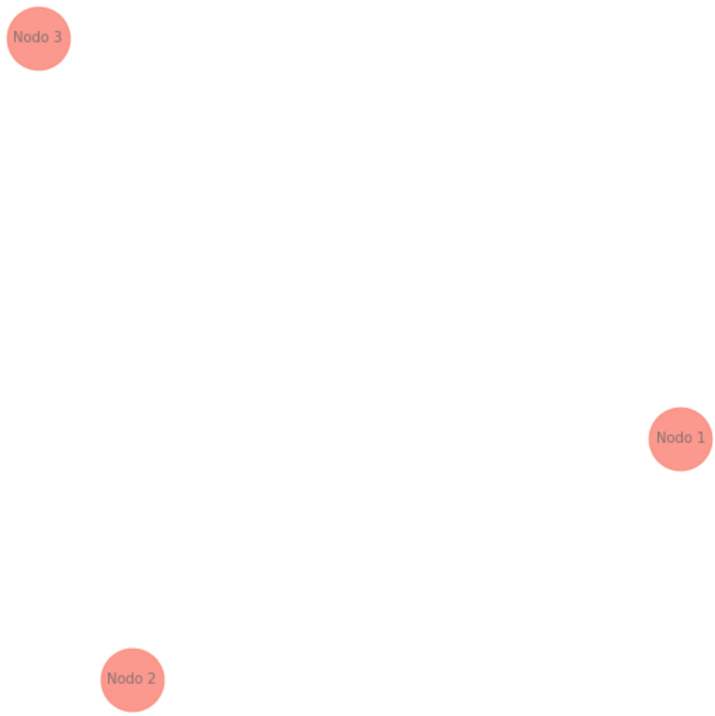
node_color: *Cambiamos el color del nodo.*

alpha: *le damos transparencia.*

font_size: *Tamaño de la etiqueta.*

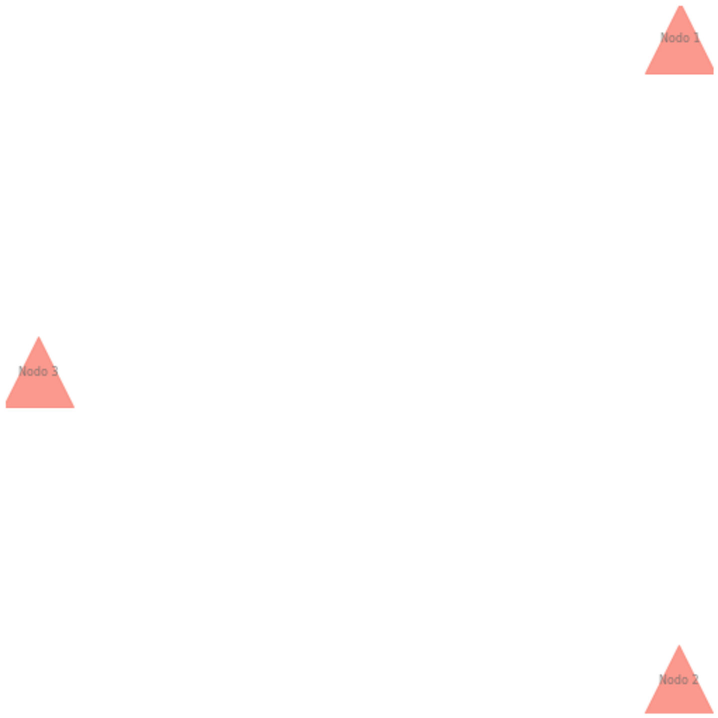
font_color: *Color de la fuente de la etiqueta.*

In [15]:
```
plt.rcParams["figure.figsize"] = [7,7]
G = nx.Graph()
G.add_node('Nodo 1')
G.add_node('Nodo 2')
G.add_node('Nodo 3')
nx.draw(G, node_color="salmon", with_labels=True, node_size=2000, alpha=0.8, font_size=10, font_color='dimgrey')
```



node_shape: *Permite cambiar la forma del nodo*

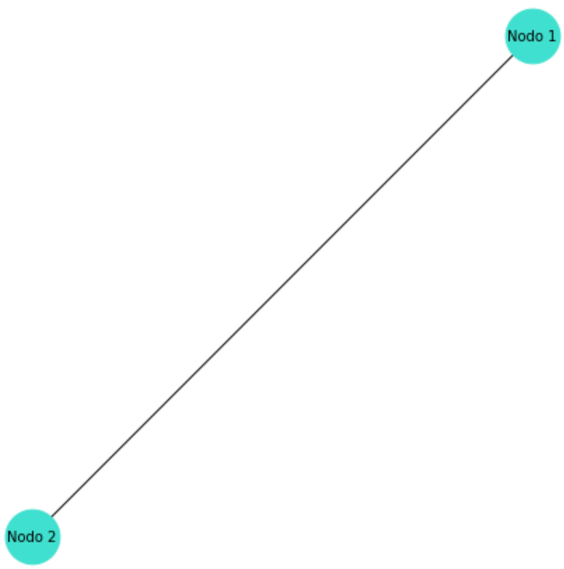
In [16]:
```
nx.draw(G, node_color="salmon", with_labels=True, node_size=2500, alpha=0.8, font_size=8,
        font_color='dimgrey', node_shape='^')
```



Aristas

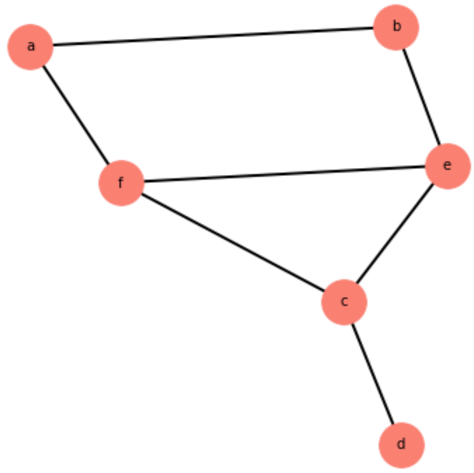
add_edge: *Enlaza nodos*.

In [17]:
```
plt.rcParams["figure.figsize"] = [6,6]
G = nx.Graph()
G.add_node('Nodo 1')
G.add_node('Nodo 2')
G.add_edge('Nodo 1','Nodo 2')
nx.draw(G, node_color="turquoise", with_labels=True, node_size=1500, font_size=10)
```



add_edges_from: *Enlaza nodos desde una lista de tuplas.*

In [18]:
```python
plt.rcParams["figure.figsize"] = [5,5]
L2 = [('a','b'),('a','f'),('b','e'),('c','e'),('c','d'),('e','f'),('f','c')]
G = nx.Graph()
G.add_edges_from(L2)

nx.draw(G, node_color="salmon", font_size=10, width=2, with_labels=True, node_size=1000)
```



Eliminar enlaces

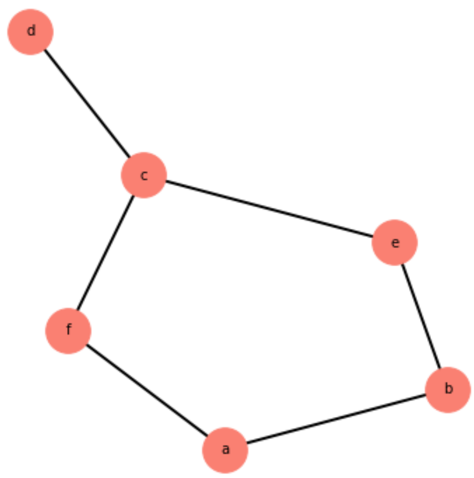
In [19]:
```python
print('Revisamos nodos y enlaces:\n')
list(G.nodes), list(G.edges)
```

```
Revisamos nodos y enlaces:
```

Out[19]:
```
([ 'a', 'b', 'f', 'e', 'c', 'd'],
 [('a', 'b'),
  ('a', 'f'),
  ('b', 'e'),
  ('f', 'e'),
  ('f', 'c'),
  ('e', 'c'),
  ('c', 'd')])
```

remove_edge: *se eliminan los enlaces.*

In [20]:
```python
plt.rcParams["figure.figsize"] = [5,5]
G.remove_edge('f','e')
nx.draw(G, node_color="salmon", font_size=10, width=2, with_labels=True, node_size=1000)
```



In [21]:
```
print('Volvemos a revisar nodos y enlaces:\n')
list(G.nodes), list(G.edges)
```

```
Volvemos a revisar nodos y enlaces:
```

Out[21]:
```
(['a', 'b', 'f', 'e', 'c', 'd'],
 [('a', 'b'), ('a', 'f'), ('b', 'e'), ('f', 'c'), ('e', 'c'), ('c', 'd')])
```

Atributos de los enlaces o aristas

edges()

In [22]:
```
print("Enlaces: ", G.edges())
```

```
Enlaces:  [('a', 'b'), ('a', 'f'), ('b', 'e'), ('f', 'c'), ('e', 'c'), ('c', 'd')]
```

number_of_edges

In [23]:
```
print("Número de enlaces: ", G.number_of_edges())
```

```
Número de enlaces:  6
```

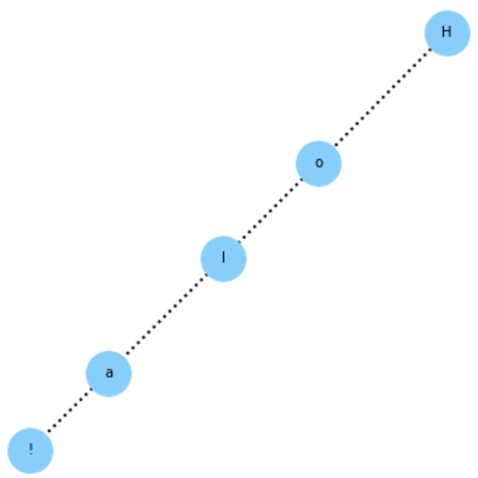
data: *Los atributos de un enlace también se devuelven como listas de tuplas compuestas de pares de nodos y un diccionario de sus atributos.*

In [24]:
```
G = nx.Graph()
G.add_node('Nodo 1')
G.add_node('Nodo 2')
G.add_node('Nodo 3')
G.add_edge('Nodo 1','Nodo 2', weight=5.0)
G.add_edge('Nodo 2','Nodo 3', weight=3.5, capacity=15, length=342.7)
G.add_edge('Nodo 3','Nodo 1', weight=1.5)
list(G.edges(data=True)), list(G.edges.data())
```

Out[24]:
```
([('Nodo\xa01', 'Nodo\xa02', {'weight': 5.0}),
  ('Nodo\xa01', 'Nodo\xa03', {'weight': 1.5}),
  ('Nodo\xa02',
   'Nodo\xa03',
   {'weight': 3.5, 'capacity': 15, 'length': 342.7})],
 [('Nodo\xa01', 'Nodo\xa02', {'weight': 5.0}),
  ('Nodo\xa01', 'Nodo\xa03', {'weight': 1.5}),
  ('Nodo\xa02',
   'Nodo\xa03',
   {'weight': 3.5, 'capacity': 15, 'length': 342.7})])
```

weight: *Agregamos peso a los enlaces.*

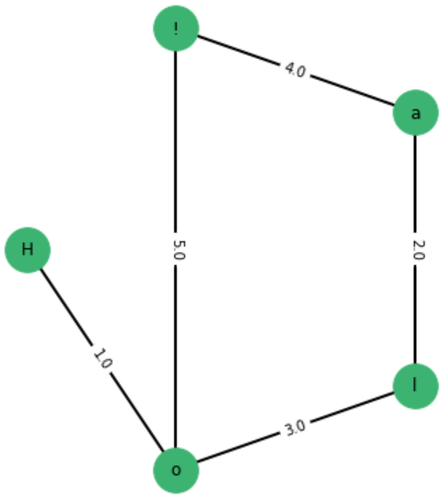
In [25]:
```
plt.rcParams["figure.figsize"] = [5,5]
G = nx.Graph()
G.add_nodes_from("Hola!")
G.add_edge('H','o', weight=1.0)
G.add_edge('o','l', weight=3.0)
G.add_edge('l','a', weight=2.0)
G.add_edge('a','!', weight=4.0)
nx.draw(G, node_color="lightskyblue", font_size=10, width=2, with_labels=True,node_size=1000,style='dotted')
```



edge_labels: *Podemos mostrar los pesos de los enlaces*

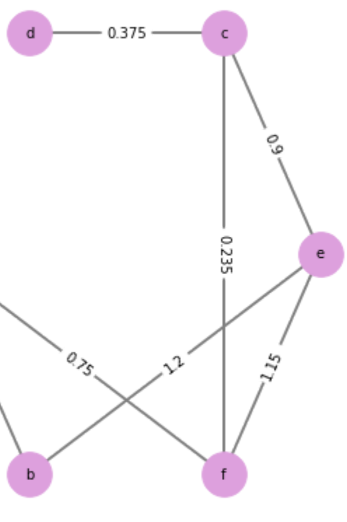
In [26]:

```python
plt.rcParams["figure.figsize"] = [6,7]
G = nx.Graph()
G.add_nodes_from("Hola!")
G.add_edge('H','o', weight=1.0)
G.add_edge('o','l', weight=3.0)
G.add_edge('l','a', weight=2.0)
G.add_edge('a','!', weight=4.0)
G.add_edge('o','!', weight=5.0)
pos = nx.shell_layout(G)
nx.draw_networkx_nodes(G, pos, node_color="mediumseagreen", node_size=1000) # nodos
nx.draw_networkx_labels(G, pos, font_size=12, font_family='sans-serif') # etiquetas de los nodos
nx.draw_networkx_edges(G, pos, width=2) # enlaces
labels = nx.get_edge_attributes(G, 'weight') # pesos de los enlaces

nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.axis('off')
plt.show()
```



In [27]:

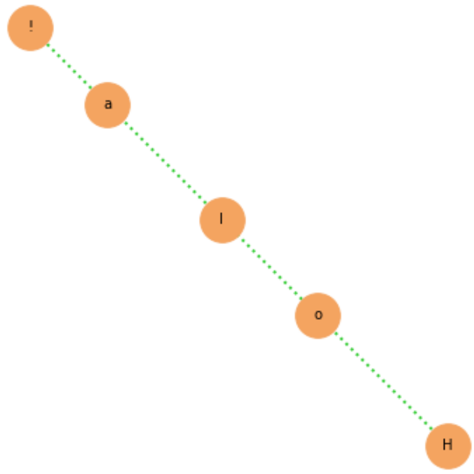
```python
plt.rcParams["figure.figsize"] = [6,7]

def emitoGraph(G, pos):
    nx.draw_networkx_nodes(G, pos, node_color = "plum", node_size = 1000)
    nx.draw_networkx_labels(G, pos, font_size = 10, font_family = 'sans-serif')
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edges(G, pos, edge_color='gray', width=2, arrowstyle= '<|-|>', arrowsize = 20)
    nx.draw_networkx_edge_labels(G, pos, edge_labels = labels)

def cargoGraph(G):
    G.add_weighted_edges_from([('a','b',0.125),('a','f',0.75),('b','e',1.2),
                               ('c','e',0.90),('c','d',0.375),('e','f',1.15),('f','c',0.235)])
G = nx.Graph()
cargoGraph(G)
pos = nx.shell_layout(G)
emitoGraph(G, pos)
plt.axis('off')
plt.show()
```



edge_color: *Agregamos color*

style: *damos un estilo*

In [28]:
```
plt.rcParams["figure.figsize"] = [5,5]
G = nx.Graph()
G.add_nodes_from("Hola!")
G.add_edge('H','o', weight=1.0)
G.add_edge('o','l', weight=3.0)
G.add_edge('l','a', weight=2.0)
G.add_edge('a','!', weight=4.0)
nx.draw(G, node_color="sandybrown", edge_color="limegreen", font_size=10, width=2, with_labels=True,
        node_size=1000, style='dotted')
```

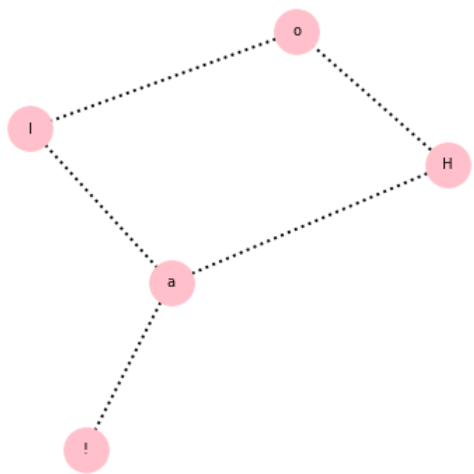


Consultar el grafo

neighbors: *Podemos ver los vecinos con neighbors().*

In [29]:
```
plt.rcParams["figure.figsize"] = [5,5]
G = nx.Graph()
G.add_nodes_from("Hola!")
G.add_edge('H','o')
G.add_edge('o','l')
G.add_edge('l','a')
G.add_edge('a','!')
G.add_edge('a','H')
nx.draw(G, node_color="pink", edge_color="black", font_size=10, width=2, with_labels=True,
        node_size=1000, style='dotted')
print("Vecinos: ", list(G.neighbors('o')))
```

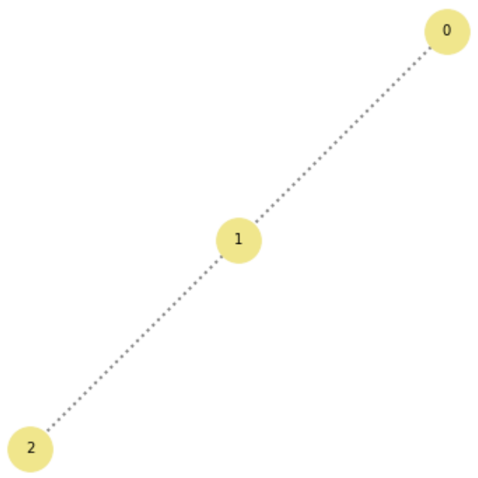
```
Vecinos:  ['H', 'l']
```



path_graph(): *devuelve el grafo de ruta n nodos, conectados linealmente por n-1 aristas. Las etiquetas de nodo son los enteros 0 a n - 1.*

In [30]:

```
G=nx.path_graph(3)
nx.draw(G, node_color="khaki", edge_color="gray", font_size=10, width=2, with_labels=True, node_size=1000,
        style='dotted')
```



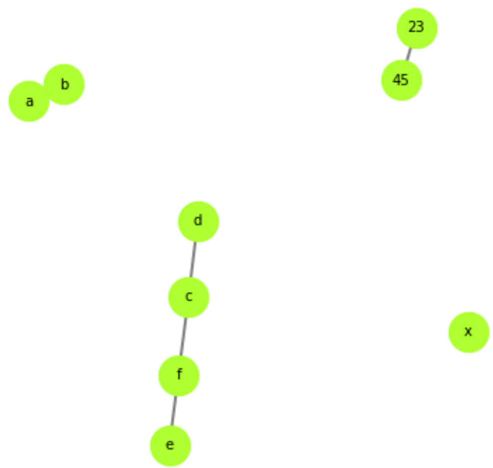
Los nodos pueden ser asignados desde listas con add_nodes_from -agrega múltiples nodos- y los enlaces pueden ser asignados por listas de tuplas con add_edges_from -agrega múltiples enlaces.

size()

In [31]:

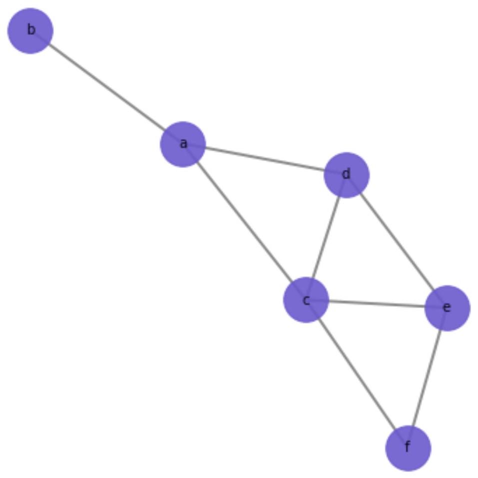
```
plt.rcParams["figure.autolayout"] = True
L1 = ['a', 'b', 'c', 'd', 'e', 'f']
L2 = [('a', 'b'),('c', 'd'), ('e', 'f'), ('f', 'c')]
G = nx.Graph()
G.add_nodes_from(L1)
G.add_edges_from(L2)
G.add_node('x')
G.add_edge(23,45)
G.nodes()
G.edges()
G.degree('a')
G.degree()
nx.draw(G, node_color="greenyellow", edge_color="gray", font_size=10, width=2, with_labels=True, node_size=800)

print("Número de aristas", G.size())
```

```
Número de aristas 5
```



Grafo Simple: G es simple si y sólo si **no** tiene aristas paralelas ni bucles:

In [32]:
```
L1 = ['a', 'b', 'c', 'd', 'e', 'f']
L2 = [('a', 'b'), ('a', 'c'), ('c', 'd'), ('d', 'a'),('e', 'c'), ('e', 'd'), ('e', 'f'), ('f', 'c')]
G = nx.Graph()
G.add_nodes_from(L1)
G.add_edges_from(L2)
nx.draw(G, node_color="slateblue", edge_color="gray", font_size=10, width=2, with_labels=True, node_size=1000, alp
```



order()

In [33]:
```
print("Número de vértices", G.order())
```

```
Número de vértices 6
```

len(G)

In [34]:
```
print("Número de vértices", len(G))
```

```
Número de vértices 6
```

degree

Grado o valencia: Sea el grafo G = (V , A , Φ). Función grado: g: V → N_0. Dónde g(v_i) = cantidad de aristas incidentes en v_i, los bucles se cuentan dobles.

In [35]:
```
print("Número de vértices", G.degree())
```

```
Número de vértices [('a', 3), ('b', 1), ('c', 4), ('d', 3), ('e', 3), ('f', 2)]
```

In [36]:
```
print("Número de vértices", G.degree('c'))
```

```
Número de vértices 4
```

In [37]:
```
print("Cantidad de vértices adyacentes a 'b'", G.degree['b'])
```

```
Cantidad de vértices adyacentes a 'b' 1
```

El grado del nodo es el número de aristas adyacentes al nodo. El grado de nodo ponderado es la suma de los pesos de borde para los bordes que inciden en ese nodo.

Propiedad: En todo grafo se cumple que la suma de los grados de los vértices es igual al doble de la cantidad de aristas. En símbolos: $\sum g(v_i) = 2 | A |$

Ejercicio resuelto: ¿Cuál es la cantidad total de vértices de un grafo que tiene 2 vértices de grado 4, 1 de grado 3, 5 de grado 2 y el resto colgantes (de grado 1) sabiendo que en total hay 12 aristas?

Usando la propiedad anterior: 2 * 4 + 1 * 3 + 5 * 2 + x * 1 = 2 * 12

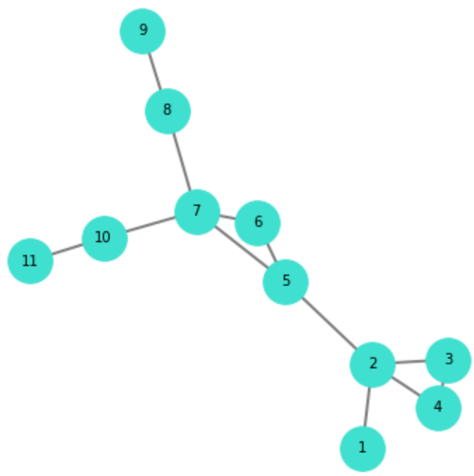
21 + x = 24 => x = 3 (cantidad de vértices colgantes)

Total de vértices: |V| = 2 + 1 + 5 + 3 = 11

Una forma posible de dibujarlo:

In [38]:
```
L1 = [1,2,3,4,5,6,7,8,9,10,11]
L2 = [(1,2),(2,3),(2,4),(2,5),(3,4),(5,6),(5,7),(6,7),(7,8),(7,10),(8,9),(10,11)]
G = nx.Graph()
pos = nx.spring_layout(G)
G.add_nodes_from(L1)
G.add_edges_from(L2)
nx.draw(G, node_color="turquoise", edge_color="gray", font_size=10, width=2, with_labels=True, node_size=1000)
G.order()
```

Out[38]: 11

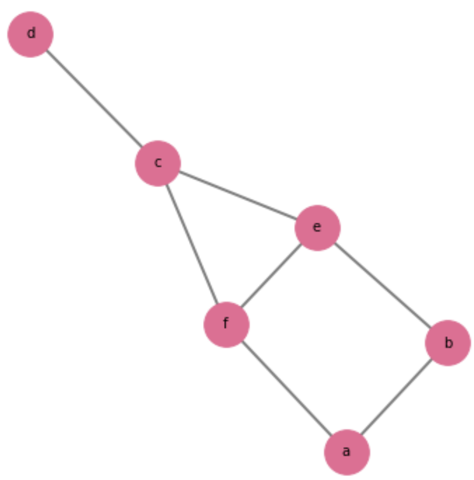


adj: *Obtiene el objeto de adyacencia que contiene los vecinos de cada nodo.*

Vértices adyacentes: v_i es adyacente a $v_j \Leftrightarrow \exists(a_k) = \{v_i,v_j\}$. Significa que son vértices que están unidos por alguna arista.

In [39]:
```
L1 = ['a', 'b', 'c', 'd', 'e', 'f']
L2 = [('a', 'b'),('a', 'f'),('b', 'e'),('c', 'e'),('c', 'd'),('e', 'f'),('f', 'c')]
G = nx.Graph()
G.add_nodes_from(L1)
G.add_edges_from(L2)
nx.draw(G, node_color="palevioletred", edge_color="gray", font_size=10, width=2, with_labels=True, node_size=1000)

print("Vértices adyacentes a 'f'", list(G.adj['f']))
```

```
Vértices adyacentes a 'f' ['a', 'e', 'c']
```



Aristas adyacentes: a_i es paralela a a_k ⇔ | Φ (a_i) ∩ Φ (a_k) | = 1. Significa que son aristas que tienen un único vértice en común.

In [40]:
```
print("Aristas adyacentes en 'd': ",G.degree["d"])
```

```
Aristas adyacentes en 'd':  1
```

Aristas incidentes en un vértice: a_i es incidente a v_k ⇔ v_k ∈ Φ (a_i). Significa que son las aristas que tienen a dicho vértice por extremo.

In [41]:
```
print("Aristas incidentes en 'c': ",G.edges('c'))
```

```
Aristas incidentes en 'c':  [('c', 'e'), ('c', 'd'), ('c', 'f')]
```

In [42]:
```
print("Valor del nodo c:\n", G['c'])
```

```
Valor del nodo c:
 {'e': {}, 'd': {}, 'f': {}}
```

adjacency_matrix: *Devuelve la matriz de adyacencia de G.*

Matriz de adyacencia: matriz booleana de n x n, Ma(G), cuyos elementos m_{ij} son 1 si v_i es adyacente a v_j, 0 si v_i no es adyacente a v_j. Significa que la matriz de adyacencia es una matriz cuadrada, las filas y las columnas representan los vértices, y los valores de los elementos son 1 si ambos vértices son adyacentes, y valen 0 en caso de no serlo.

In [43]:
```
Ma = nx.adjacency_matrix(G)
Ma.todense()
```

Out[43]:
```
matrix([[0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 1, 0],
        [0, 0, 0, 1, 1, 1],
        [0, 0, 1, 0, 0, 0],
        [0, 1, 1, 0, 0, 1],
        [1, 0, 1, 0, 1, 0]], dtype=int32)
```

Networkx almacena las matrices de forma dispersa (posiciones no nulas y valor en éstas), y nosotros estamos acostumbrados a la forma densa (forma matricial).

G.adjacency(): *Devuelve un iterador sobre (nodo, diccionario de adyacencia) para todos los nodos del gráfico.*

In [44]:
```
G = nx.path_graph(6)  # or DiGraph, MultiGraph, MultiDiGraph, etc
[(n, nbrdict) for n, nbrdict in G.adjacency()]
```

Out[44]:
```
[(0, {1: {}}),
 (1, {0: {}, 2: {}}),
 (2, {1: {}, 3: {}}),
 (3, {2: {}, 4: {}}),
 (4, {3: {}, 5: {}}),
 (5, {4: {}})]
```

incidence_matrix

Matriz de incidencia: matriz booleana de n x n, Mi(G), cuyos elementos m_{ij} son 1 si v_i es extremo de a_j, 0 si v_i no es extremo de a v_j. Significa que la matriz de incidencia es una matriz rectangular, las filas representan los vértices, y las columnas representan las aristas, y los valores de los elementos son 1 si el vértice es extremo de la arista, y valen 0 en caso de no serlo.

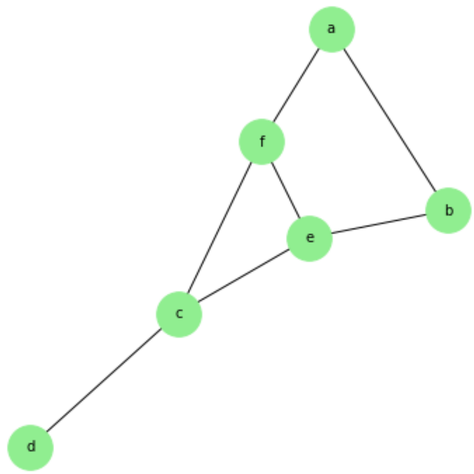
In [45]:
```
Mi = nx.incidence_matrix(G)
Mi.todense()
```

Out[45]:
```
array([[1., 0., 0., 0., 0.],
       [1., 1., 0., 0., 0.],
       [0., 1., 1., 0., 0.],
       [0., 0., 1., 1., 0.],
       [0., 0., 0., 1., 1.],
       [0., 0., 0., 0., 1.]])
```

G.adj.items(): *El examen de todos los pares (nodo, adyacencia) se logra usando adj.items() o adjacency(). Para los grafos no dirigidos, la iteración de adyacencia ve cada borde dos veces. Los enlaces se deben dar como tuplas de 3 (u, v, w) donde w es un número.*

In [46]:

```
plt.rcParams["figure.autolayout"] = True
G = nx.Graph()
G.add_weighted_edges_from([('a','b',0.125),('a','f',0.75),('b','e',1.2),
                           ('c','e',0.90),('c','d',0.375),('e','f',1.15),
                           ('f','c',0.235)])
nx.draw(G, node_color="lightgreen", edge_color="black", font_size=10, width=1, with_labels=True, node_size=1000)
```



In [47]:

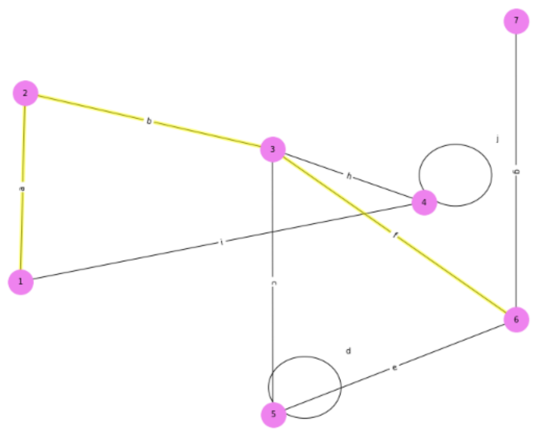
```
for n, nbrs in G.adj.items():
    print(f"\n(Nodo: {n}, Enlaces: {nbrs})\n")

    for nbr, eattr in nbrs.items():
        print(f"(Enlace: {nbr}, Atributo: {eattr})")

for (u, v, wt) in G.edges.data('weight'):
    print(f"({u}, {v}, {wt})")
```

```
(Nodo: a, Enlaces: {'b': {'weight': 0.125}, 'f': {'weight': 0.75}})

(Enlace: b, Atributo: {'weight': 0.125})
(Enlace: f, Atributo: {'weight': 0.75})

(Nodo: b, Enlaces: {'a': {'weight': 0.125}, 'e': {'weight': 1.2}})

(Enlace: a, Atributo: {'weight': 0.125})
(Enlace: e, Atributo: {'weight': 1.2})

(Nodo: f, Enlaces: {'a': {'weight': 0.75}, 'e': {'weight': 1.15}, 'c': {'weight': 0.235}})

(Enlace: a, Atributo: {'weight': 0.75})
(Enlace: e, Atributo: {'weight': 1.15})
(Enlace: c, Atributo: {'weight': 0.235})

(Nodo: e, Enlaces: {'b': {'weight': 1.2}, 'c': {'weight': 0.9}, 'f': {'weight': 1.15}})

(Enlace: b, Atributo: {'weight': 1.2})
(Enlace: c, Atributo: {'weight': 0.9})
(Enlace: f, Atributo: {'weight': 1.15})

(Nodo: c, Enlaces: {'e': {'weight': 0.9}, 'd': {'weight': 0.375}, 'f': {'weight': 0.235}})

(Enlace: e, Atributo: {'weight': 0.9})
(Enlace: d, Atributo: {'weight': 0.375})
(Enlace: f, Atributo: {'weight': 0.235})

(Nodo: d, Enlaces: {'c': {'weight': 0.375}})

(Enlace: c, Atributo: {'weight': 0.375})
(a, b, 0.125)
(a, f, 0.75)
(b, e, 1.2)
(f, e, 1.15)
(f, c, 0.235)
(e, c, 0.9)
(c, d, 0.375)
```

Caminos y ciclos en grafos

Camino: sucesión de aristas adyacentes distintas.

```
Un posible camino es:     C1= ( 1, a, 2, b, 3, f, 6 )
```



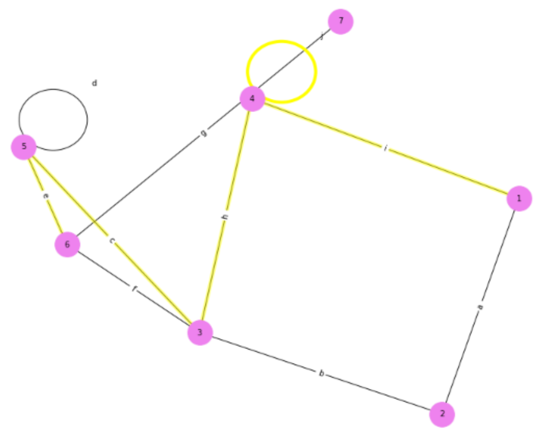
```
La longitud de este camino es: long[C1] = 3
Es un camino simple porque no repite vértices
```

Longitud de un camino: cantidad de aristas que lo componen.

Camino simple: si todos los vértices son distintos.

Camino elemental: si todas las aristas son distintas.

```
Otro posible camino es:     C2 = ( 1, i, 4, j, 4, h, 3, c, 5, e, 6 )
```



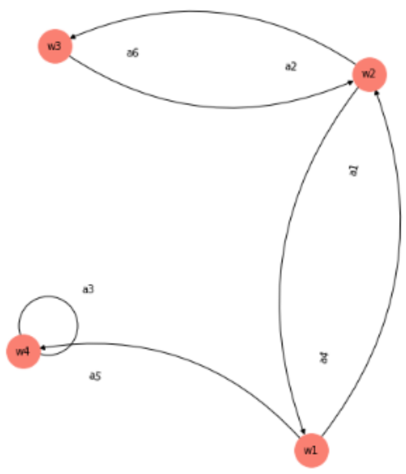
```
La longitud de este camino es: long[C2] = 5
Este camino NO es simple porque repite el vértice 4.
```

V = { w1, w2, w3 , w4 } A = { a1, a2, a3, a4, a5, a6 }

$_\delta$(a1)=(w1,w2) $_\delta$(a2)=(w2,w3) $_\delta$(a3)=(w4,w4)

$_\delta$(a4)=(w2,w1) $_\delta$(a5)=(w4,w1) $_\delta$(a6)= (w2,w3)

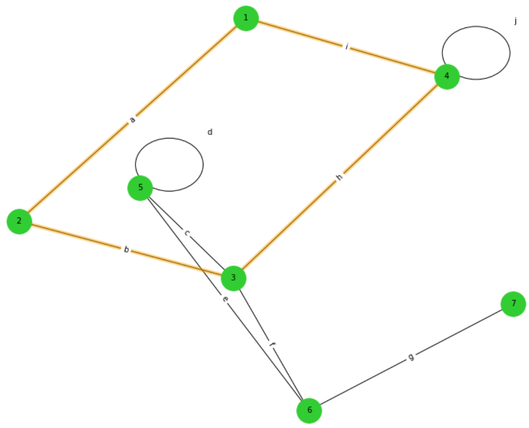
```
Se puede diagramar de la siguiente forma:
```




```
Extremo inicial de a5: w4
Extremo final de a5:  w1
Buble: a3
Aristas Paralelas:     a2 y a6
Aristas Antiparalelas:  a1 y a4
Camino: C = ( w4, a5, w1, a1, w2, a2, w3)
```

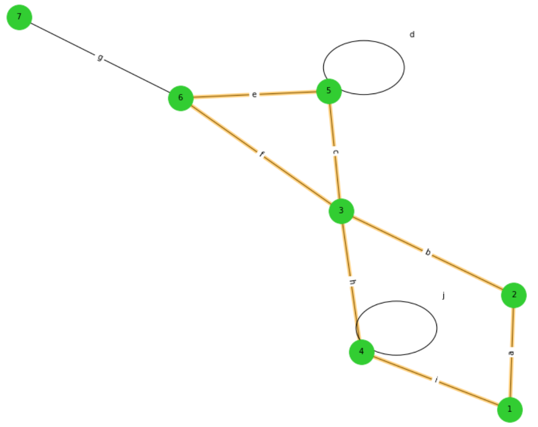
Ciclo o circuito: camino cerrado (vértice inicial = vértice final)

```
Un posible ciclo es:     C1 = ( 1, a, 2, b, 3, h, 4, i, 1)
```



```
La longitud de este ciclo es: long[C1] = 4
Este ciclo es simple pues no repite vértices.
```

```
Otro posible ciclo es:     C3 = ( 1, a, 2, b, 3, c, 5, e, 6, f, 3, h, 4, i, 1 )
```



```
La longitud de este ciclo es: long[C3] = 7
Este ciclo NO es simple porque repite el vértice 3.
```

Caminos y ciclos especiales

Caminos y ciclos eulerianos

Camino de Euler: camino que pasa por todas las aristas.

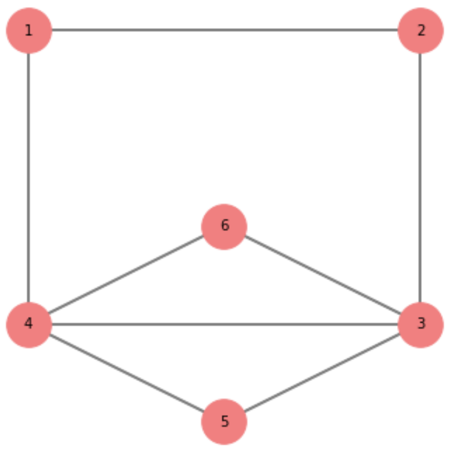
```
La condición necesaria y suficiente para que en un grafo exista camino euleriano es: El grafo debe ser conex
o, y todos
los vértices deben tener grado par, o a lo sumo dos grados impar.
```

Ciclo de Euler: Ciclo que pasa por todas las aristas del grafo.

```
La condición necesaria y suficiente para que en un grafo exista ciclo euleriano es: El grafo debe ser conexo,
y todos
los vértices deben tener grado par.
```

In [48]:

```
plt.rcParams["figure.autolayout"] = True
fig, ax = plt.subplots()
G = nx.Graph([(1,2),(2,3),(3,4),(4,6),(6,3),(3,5),(5,4),(4,1)])
pos_dict = {1:[ -0.1,1.4], 2:[0.1, 1.4],3: [ 0.1,0.8], 4: [-0.1,0.8], 6:[0.,1.], 5: [0.,0.6]}
positions=nx.spring_layout(G, pos=pos_dict)
nx.draw(G, pos_dict, with_labels=True, node_color="lightcoral", font_size=10, width=1, node_size=1000)
nx.draw_networkx_edges(G, pos_dict, edge_color='grey', width=2)
plt.show()
```




```
C = {1,2,3,4,6,3,5,4,1} es un ciclo euleriano.
```

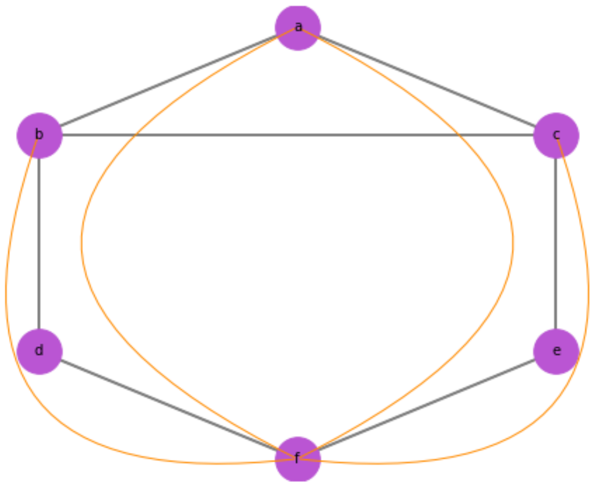
Caminos y ciclos hamiltonianos

Camino de Hamilton: camino simple que pasa por todos los vértices.

Ciclo de Hamilton: Ciclo simple que pasa por todos los vértices.

Observación: no necesariamente va a pasar por todas las aristas, pues en muchos casos repetiría vértices y no sería hamiltoniano.

Un posible grafo hamiltoniano es: (A, B, D, F, E, C, A)



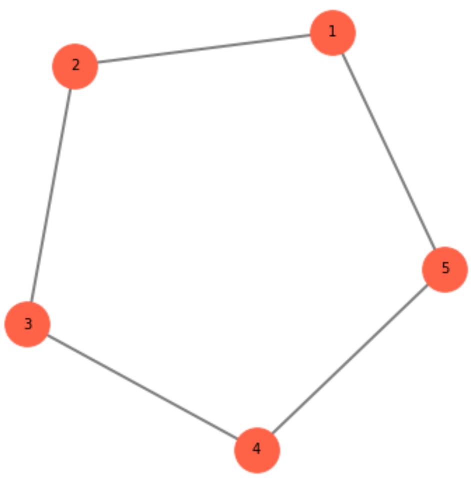
Grafos regulares

Grafo K-regular: G es k-regular ⇔ ∀ v ∈ V : g(v) = k con k ∈ N_0

a_1 y a_5 son paralelas, a_1 es paralela a a_k Φ (a_i) = (a_k). Significa que son aristas comprendidas entre los mismos vértices.

El siguiente grafo es 2-regular pues todos los vértices tienen grado 2.

In [49]:
```
plt.rcParams["figure.autolayout"] = True
L1 = [1,2,3,4,5]
L2 = [(1,2),(2,3),(3,4),(4,5),(5,1)]
G = nx.Graph()
pos = nx.random_layout(G)
G.add_nodes_from(L1)
G.add_edges_from(L2)
nx.draw(G, node_color="tomato", edge_color="gray", font_size=10, width=2, with_labels=True, node_size=1000)
```



Isomorfismos de grafos

Dados 2 grafos: $G_1 = (V_1 , A_1 , \Phi_1)$ y $G_2 = (V_2 , A_2 , \Phi_2)$.

Se dice que son isomorfos si y solo si existen dos funciones biyectivas: $f: V_1 \rightarrow V_2$ y $g: A_1 \rightarrow A_2$

Tales que: $\forall\, a \in A_1 : \Phi_2(g(a)) = f(\Phi_1(a))$

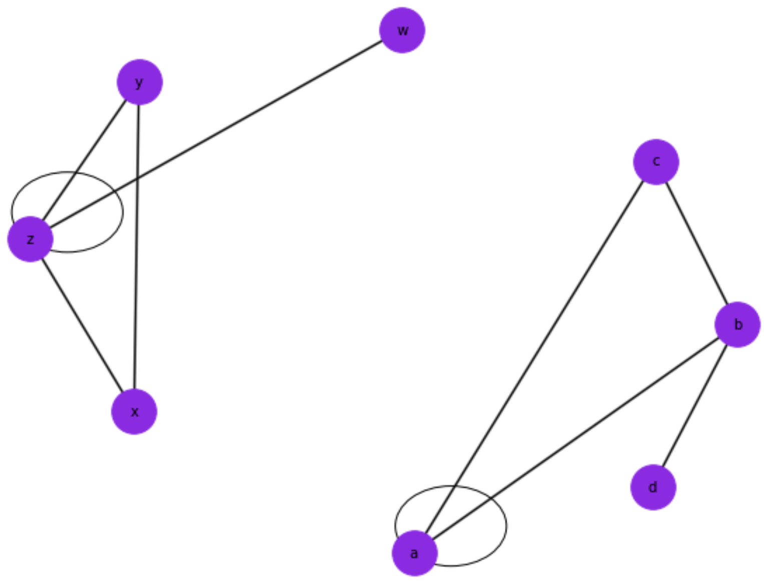
Si no hay aristas paralelas, entonces es suficiente: $\forall\, u,v \in V_1: \{u,v\} \in A_1 \rightarrow \{f(u), f(v)\} \in A_2$

Esto significa que si en el primer grafo hay una arista entre dos vértices, los correspondientes a estos vértices
en el segundo grafo también deben estar unidos por una arista.

En pocas palabras, dos grafos son isomorfos cuando tienen la misma estructura, es decir sus vértices están relacionados de igual forma aunque estén dibujados de manera distinta.

Condiciones necesarias para que 2 grafos sean isomorfos:
- Deben tener la misma cantidad de vértices.
- Deben tener la misma cantidad de aristas.
- Deben tener los mismos grados de los vértices.
- Deben tener caminos de las mismas longitudes.
- Si uno tiene ciclos, el otro también debe tenerlos.

Observación: las condiciones mencionadas son necesarias (es decir que sí o sí se deben cumplir para que los grafos
sean isomorfos) pero no son suficientes (o sea que aunque se cumplan puede ser que los grafos no sean isomorfos).
Para estar seguros que dos grafos son isomorfos, una condición que es suficiente es que tengan la misma matriz de
adyacencia.



Analicemos si los grafos anteriores son isomorfos:
- G1 = {a,b,c,d}
- G2 = {w,x,y,z}

Ambos tienen 4 vértices y 5 aristas. La función biyectiva, haciendo corresponder los vértices con iguales grados:
f(a) = y ; f(b) = z ; f(c) = x ; f(d) = w
En la definición decía que si entre dos vértices del primer grafo había una arista, también debía haber arista entre los vértices correspondientes en el segundo grafo.

Por ejemplo entre a y b hay una arista en G1, y también hay una arista entre f(a) y f(b) en G2.
Esto mismo habría que revisar para cada arista, ello se puede hacer todo junto con la matriz ordenando convenientemente los vértices:

G1

	a	b	c	d
a	1	1	1	0
b	1	0	1	1
c	1	1	0	0
d	0	1	0	0

G2

	y	z	x	w
y	1	1	1	0
z	1	0	1	1
x	1	1	0	0
w	0	1	0	0

Como las matrices son iguales podemos asegurar que G1 es isomorfo a G2.

Digrafo

Definición formal: Un digrafo es una terna G = (V , A , Φ)

Dónde:

V: Conjuntos de vértices, dónde V ≠ ∅

A: Conjuntos de aristas dirigidas.

Φ: Función de incidencia Φ: A → VXV

La función de incidencia Φ le hace corresponder a cada arista un par ordenado de vértices, al primero se lo llama extremo inicial de la arista, y el segundo es el vértice final.

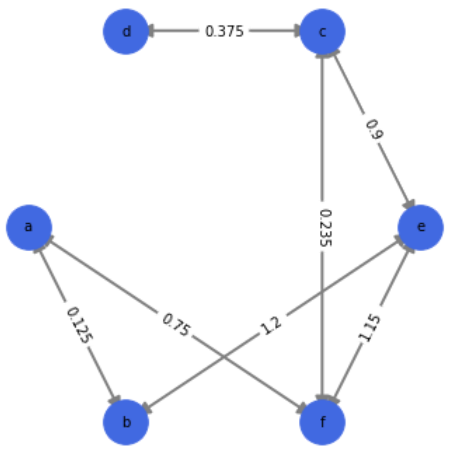
Los caminos y los ciclos se definen de la misma forma que para los grafos no dirigidos, pero hay que respetar el sentido de las aristas.

DiGraph()

to_directed()

In [50]:

```
def emitoGraph(G, pos):
    nx.draw_networkx_nodes(G, pos, node_color = "royalblue", node_size = 1000)
    nx.draw_networkx_labels(G, pos, font_size = 10, font_family = 'sans-serif')
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edges(G, pos, edge_color='gray', width=2, arrowstyle= '<|-|>', arrowsize = 20)
    nx.draw_networkx_edge_labels(G, pos, edge_labels = labels)

def cargoGraph(G):
    G.add_weighted_edges_from([('a','b',0.125),('a','f',0.75),('b','e',1.2),
                               ('c','e',0.90),('c','d',0.375),('e','f',1.15),('f','c',0.235)])
H = G.to_directed()
H = nx.DiGraph()
cargoGraph(H)
pos = nx.shell_layout(H)
emitoGraph(H, pos)
plt.axis('off')
plt.show()
```



to_directed() convierte el grafo no dirigido a uno dirigido o directamente se puede utilizar DiGraph()

Función grado de un digrafo

Grado positivo: cantidad de arcos que "entran" al vértice. Se denota g+(v)

Grado negativo: cantidad de arcos que "salen" del vértice. Se denota g-(v)

Grado total: suma de los grados positivo y negativo. Se denota g(v)

Grado neto: Diferencia entre grado positivo y negativo. Se denota $g_N(v)$

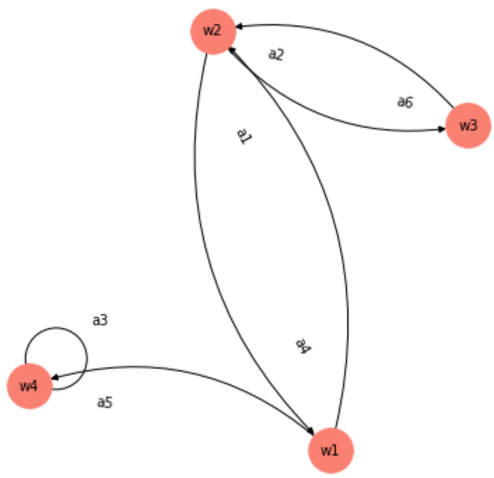
Propiedades:
$\Sigma\, g^+(v_i) = |\, A\, |$

$\Sigma\, g^-(v_i) = |\, A\, |$

$\Sigma\, g(v_i) = 2\,|\, A\, |$

$\Sigma\ g_N(v_i) = 0$

```
Ejemplo:
```



Grados positivos: $g^+(w_1) = 2$; $g^+(w_2) = 1$; $g^+(w_3) = 2$; $g^+(w_4) = 1$

Grados negativos: $g^-(w_1) = 1$; $g^-(w_2) = 3$; $g^-(w_3) = 0$; $g^-(w_4) = 2$

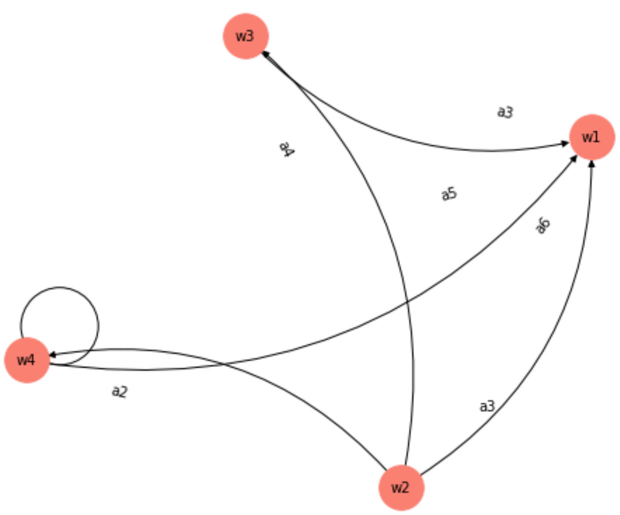
Grados totales: $g(w_1) = 3$; $g(w_2) = 4$; $g(w_3) = 2$; $g(w_4) = 3$

Grados netos: $g_N(w_1) = 1$; $g_N(w_2) = -2$; $g_N(w_3) = 2$; $g_N(w_4) = -1$

Pozo: es un vértice v tal que $g^-(v) = 0$, o sea, v no es extremo inicial de ninguna arista.

Fuente: es un vértice v tal que $g^+(v) = 0$, o sea, v no es extremo final de ninguna arista.

```
Ejemplo:
```




```
w1 es pozo, y w2 es fuente.
```

Representación matricial de digrafos

Sea un digrafo simple G = (V , A , Φ), con

- $V = \{v_1, v_2 \dots, v_m\}$
- $A = \{a_1, a_2 \dots a_m\}$

Matriz de adyacencia es una matriz booleana de n x n

Ma(G) cuyos elementos $m_{i,j}$ son:

1 si ∃ a ∈ A: δ(a) = (v_i v_j) 0 en caso contrario.

Matriz de incidencia es una matriz booleana de n x m

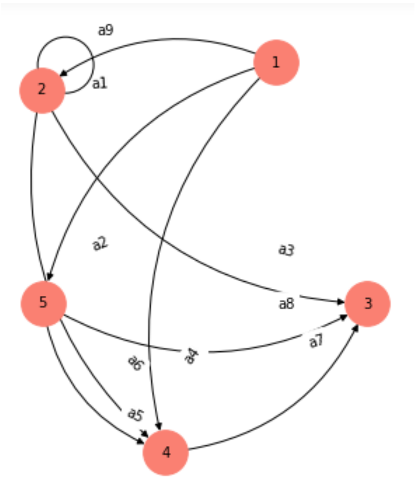
Mi(G) cuyos elementos $m_{i,j}$ son:

1 si v_i es vértice inicial de a_j

-1 si v_i es vértice final de a_j

0 si v_i no es extremo de a_j

```
Ejemplo:
```




```
Ma(DG)                                         Mi(DG)
---------------------                          --------------------------------
| 0 | 1 | 0 | 1 | 1 |                          | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |                          |-1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |                          | 0 | 0 |-1 | 0 | 0 | 0 |-1 |-1 |
| 0 | 0 | 1 | 0 | 0 |                          | 0 | 0 | 0 |-1 |-1 |-1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 |                          | 0 |-1 | 0 | 0 | 1 | 0 | 0 |-1 |
```

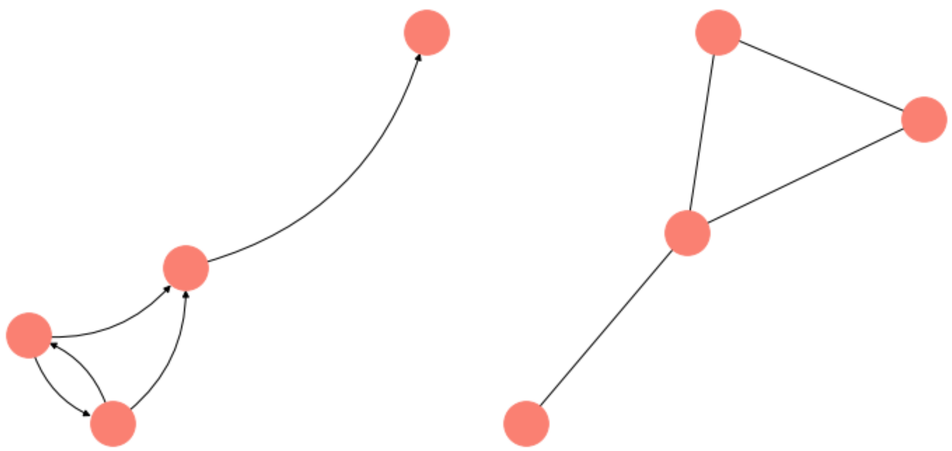
Grafo asociado a un digrafo

Dado un digrafo, si se cambian las aristas dirigidas por aristas no dirigidas, se obtiene el grafo asociado. Es decir hay que ignorar el sentido de las aristas. Si en el digrafo original hay aristas paralelas o antiparalelas, en el grafo asociado sólo se representa una de ellas.

```
Digrafo y grafo asociado
```

In [51]:

```python
plt.rcParams["figure.figsize"] = (10,5)
fig, axes = plt.subplots(nrows=1, ncols=2)
ax = axes.flatten()

DG = nx.DiGraph([('v1','v2'),('v2','v4'),('v1','v4'),('v2','v1'), ('v4', 'v3')])
pos = nx.spring_layout(DG)
nx.draw(DG,pos,with_labels=False,connectionstyle="arc3,rad=0.3",node_color="salmon",
        font_size=10, width=1, node_size=1000, ax=ax[0])
ax[0].set_axis_off()

G = nx.Graph([('v1','v2'),('v2','v4'),('v1','v4'),('v4','v3')])
pos = nx.spring_layout(G)
nx.draw(G,pos,with_labels=False,node_color="salmon",font_size=10,width=1,node_size=1000, ax=ax[1])
ax[1].set_axis_off()
```

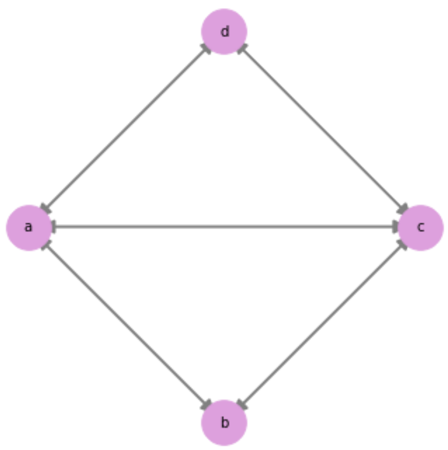


Conexidad en dígrafos

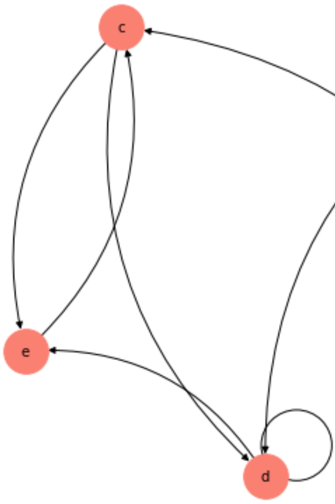
Dígrafo conexo: es todo aquel cuyo grafo asociado sea conexo.
Dígrafo fuertemente conexo: es todo aquel en el que exista algún camino entre todo par de vértices.

In [52]:

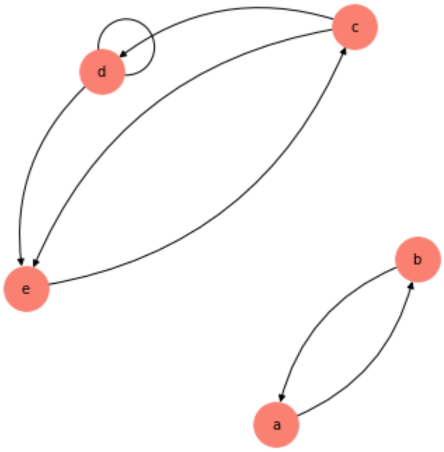
```python
plt.rcParams["figure.figsize"] = [5,5]
def emitoGraph(G, pos):
    nx.draw_networkx_nodes(G, pos, node_color = "plum", node_size = 1000)
    nx.draw_networkx_labels(G, pos, font_size = 10, font_family = 'sans-serif')
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edges(G, pos, edge_color='gray', width=2, arrowstyle= '<|-|>', arrowsize = 20)
    nx.draw_networkx_edge_labels(G, pos, edge_labels = labels)
def cargoGraph(G):
    G.add_edges_from([('a','b'),('b','c'),('c','d'),
                      ('a','d'),('a','c')])
DG = nx.DiGraph()
cargoGraph(DG)
pos = nx.shell_layout(DG)
emitoGraph(DG, pos)
plt.axis('off')
plt.show()
```



El digrafo anterior es conexo y además es fuertemente conexo.




```
Este digrafo si bien es conexo, no es fuertemente conexo, ya que por ejemplo no existe camino alguno que salg
a del
vértice C y llegue al vértice B.
```




```
Lo que sí hay son dos componentes fuertemente conexas.
```

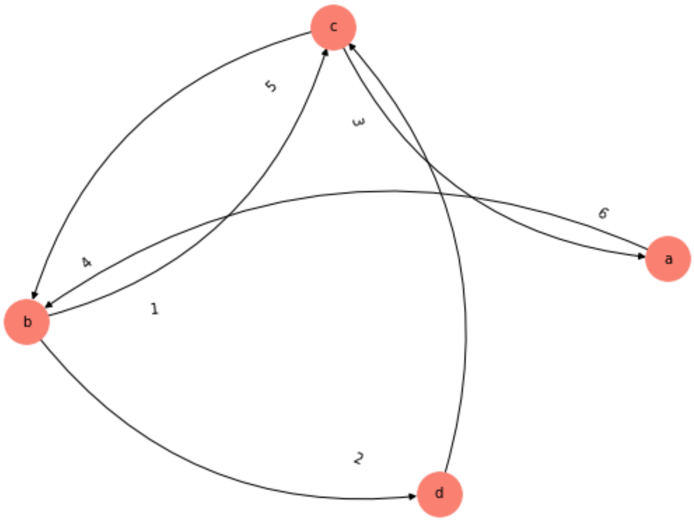
Caminos de Euler y Hamilton en digrafos

Se definen de forma similar que para grafos no dirigidos, pero hay que respetar el sentido de las aristas.

Condición necesaria y suficiente para que exista ciclo de Euler en un digrafo: $\forall$ v $\in$V : g+(v) = g-(v)

In [53]:

```
plt.rcParams["figure.figsize"] = [8, 6]
plt.rcParams["figure.autolayout"] = True
fig, ax = plt.subplots()

DG = nx.DiGraph([('a','b',{'label':'1'}),('b','d',{'label':'2'}),('d','c',{'label':'3'}), ('c','a',{'label':'6'}),
                 ('b','c',{'label':'5'}), ('c','b',{'label':'4'})])

pos = nx.spring_layout(DG,k=3,scale=2.0)
labels = nx.get_edge_attributes(DG,'label')
nx.draw(DG, pos, with_labels=True, connectionstyle="arc3,rad=0.3", node_color="salmon", font_size=10,
        width=1, node_size=1000)
nx.draw_networkx_edge_labels(DG, pos, edge_labels=labels, label_pos=0.2)
plt.show()
```

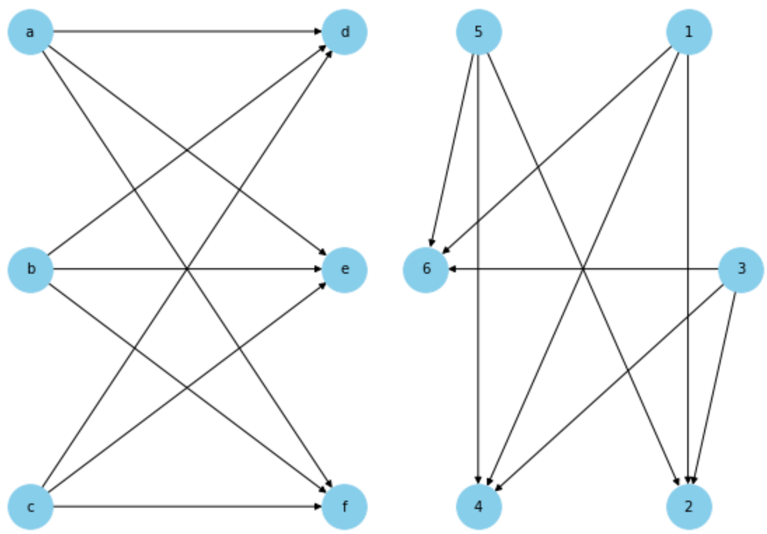



```
En este digrafo existe ciclo de Euler: C = (A,1,B,2,D,3,C,4,B,5,C,6,A)
y un posible ciclo de Hamilton:  C = (A,1,B,2,D,3,C,6,A)
```

Isomorfismos de digrafos

Es lo mismo que para grafos, pero hay que tener en cuenta el sentido de las aristas.

In [54]:
```python
fig, axes = plt.subplots(nrows=1, ncols=2)
ax = axes.flatten()

D1 = nx.DiGraph([('a','d'),('a','e'),('a','f'),('b','d'),('b','e'),('b','f'),('c','d'),('c','f'),('c','e')])
pos_dict = {'a': [-0.1, 0.9], 'b': [-0.1,0.7], 'c': [-0.1,0.5], 'd': [0.1,0.9], 'e': [0.1,0.7],'f': [0.1,0.5]}
nx.draw(D1, pos_dict, with_labels=True, node_color="skyblue", font_size=10, width=1, node_size=1000, ax=ax[0])
ax[0].set_axis_off()

D2 = nx.DiGraph([('3','4'),('3','6'),('3','2'),('5','4'),('5','6'),('1','6'),('1','2'),('1','4'),('5','2')])
pos_dict = {'5': [-0.1, 0.9], '1': [0.1,0.9], '4': [-0.1,0.5], '2': [0.1,0.5], '6': [-0.15,0.7],'3': [0.15,0.7]}
nx.draw(D2, pos_dict, with_labels=True, node_color="skyblue", font_size=10, width=1, node_size=1000, ax=ax[1])
ax[1].set_axis_off()
```



Si definimos la función: f : V_1 $_1$ V_1 tal que: f(1) = A ; f(2) = D ; f(3) = B ; f(4) = E ; f(5) = C ; f(6) = F y construimos las matrices de adyacencia:

In [55]:
```python
M1 = nx.adjacency_matrix(D1)
M1.todense()
```

Out[55]:
```
matrix([[0, 1, 1, 1, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0, 0],
        [0, 1, 1, 1, 0, 0]], dtype=int32)
```

In [56]:
```python
M2 = nx.adjacency_matrix(D2)
M2.todense()
```

Out[56]:
```
matrix([[0, 1, 1, 1, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0, 0],
        [0, 1, 1, 1, 0, 0]], dtype=int32)
```

Como las matrices son iguales entonces los digrafos son isomorfos.

Grafos bipartitos

Sea un grafo simple: G = (V , A , Φ) con V = {v_1.....v_n} y A = {a_1.....a_m}

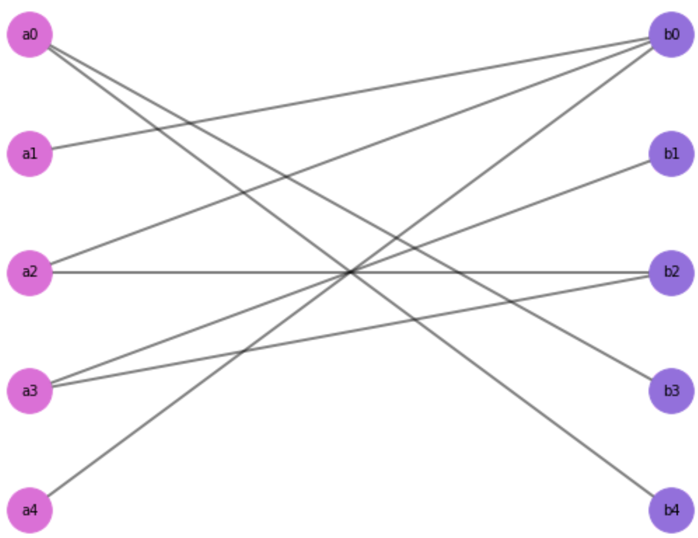
G es bipartito $\Leftrightarrow V = V_1 \cup V_2$ con $V_1 \neq \emptyset \wedge V_2 \neq \emptyset \wedge V_1 \cap V_2 = \emptyset \wedge \forall a_i \in A: \Phi(a_i) = \{v_j, v_k\}$ con $v_j \in v_1 \wedge v_k \in v$

```
Los grafos bipartitos son grafos cuyo conjunto de vértices está particionado en dos subconjuntos: V1 y V2 tal
es que los vértices de V1 pueden ser adyacentes a los vértices de V2 pero los de un mismo subconjunto no son
 adyacentes entre sí.

En el siguiente grafo, cuyo conjunto de vértices es:    V = { 1, 2, 3, 4, 5 }
Si consideramos los subconjuntos: V1 = { 1, 2, 3 }      V2 = { 4, 5 }
Vemos que todas las aristas que hay, tienen un extremo en V1 y el otro en V2, por lo tanto es bipartito.

Nota: la definición no exige que deba haber arista entre todo par de vértices (uno de V1 y el otro de V2 ) si
no que dice que las aristas que existan deben estar comprendidas entre un vértice de cada subconjunto. En est
e ejemplo, no hay
arista entre 2 y 4, lo cual estaba permitido.
```

bipartite: Los grafos bipartitos tienen dos conjuntos de nodos y bordes que solo conectan nodos de conjuntos opuestos.

In [57]:

```python
mat = [ [0, 0, 0, 1, 1],
        [1, 0, 0, 0, 0],
        [1, 0, 1, 0, 0],
        [0, 1, 1, 0, 0],
        [1, 0, 0, 0, 0]]
G = nx.Graph()
a = ["a"+str(i) for i in range(len(mat))]
b = ["b"+str(j) for j in range(len(mat[0]))]
G.add_nodes_from(a, bipartite=0)
G.add_nodes_from(b, bipartite=1)
for i in range(len(mat)):
    for j in range(len(mat[i])):
        if mat[i][j] != 0:
            G.add_edge(a[i], b[j])
pos_a = {}
const = 0.100
x = 0.100
y = 1.0
for i in range(len(a)):
    pos_a[a[i]] = [x, y-i*const]
pos_b = {}
x = 0.500
for i in range(len(b)):
    pos_b[b[i]] = [x, y-i*const]
nx.draw_networkx_nodes(G, pos_a, nodelist=a, node_color="orchid", node_size=1000)
nx.draw_networkx_nodes(G, pos_b, nodelist=b, node_color="mediumpurple", node_size=1000)
pos = {}
pos.update(pos_a)
pos.update(pos_b)
nx.draw_networkx_labels(G, pos, font_size=10)
nx.draw_networkx_edges(G, pos, width=2, alpha=0.5)
plt.axis('off')
plt.show()
```

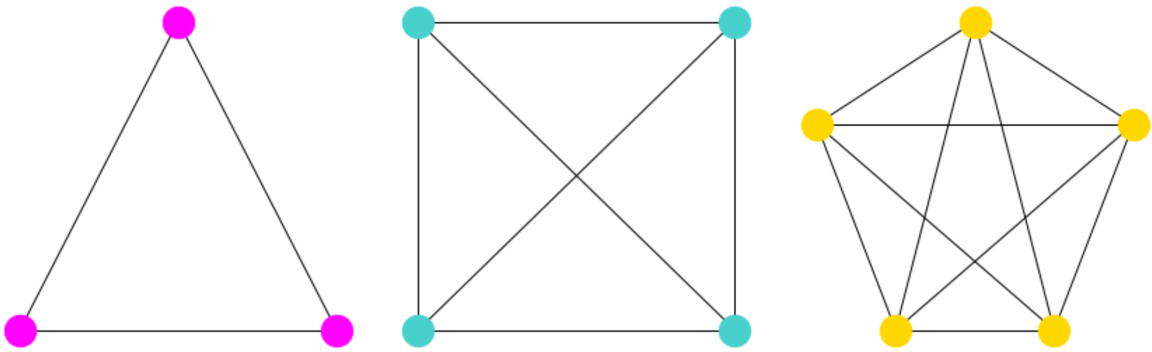


Grafos completos Kn

Sea $n \in N$: $K_n = (V, A, \Phi)$ tal que: $\forall$ v,w $\in$ V: v $\neq$ w $\Leftrightarrow \exists$ a $\in$ A: Φ (a) = {v, w}

O sea, los Kn son grafos simples de n vértices en los cuales cada vértice es adyacente a todos los demás. Ejemplos:

In [58]:
```
plt.rcParams["figure.figsize"] = [12, 4]
plt.rcParams["figure.autolayout"] = True
fig, axes = plt.subplots(nrows=1, ncols=3)
ax = axes.flatten()

G1 = nx.Graph([('a','b'),('a','c'), ('b','c')])
pos_dict = {'a':[ 0.,0.8], 'b': [-0.1,  0.5], 'c': [ 0.1,0.5]}
nx.draw(G1, pos_dict, node_color="magenta", font_size=10, width=1, node_size=500, ax=ax[0])
ax[0].set_axis_off()

G2 = nx.Graph([('a','b'),('b','c'), ('c','d'),('d','a'),('a','c'),('b','d')])
pos_dict = {'a': [-0.05, 0.8], 'b': [0.05,0.8], 'c': [-0.05,0.5], 'd': [0.05,0.5]}
nx.draw(G2, pos_dict, node_color="mediumturquoise", font_size=10, width=1, node_size=500, ax=ax[1])
ax[1].set_axis_off()

G3 = nx.Graph([('a','b'),('a','c'),('a','d'),('a','e'),('b','c'),('b','d'),('c','e'),('d','e'),('b','e'),('c','d')
pos_dict = {'a':[ 0.,1.4], 'b': [-0.1,  1.2], 'c': [ 0.1,1.2], 'd': [-0.05,0.8], 'e': [0.05,0.8]}
nx.draw(G3, pos_dict, node_color="gold", font_size=10, width=1, node_size=500, ax=ax[2])
ax[2].set_axis_off()
```



Grafos bipartitos completos $k_{n,m}$

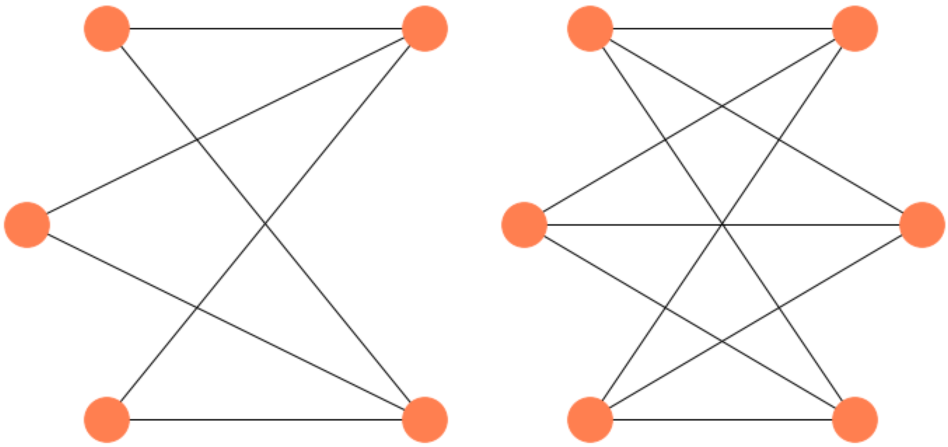
Son grafos bipartitos de n + m vértices con todas las aristas posibles. La cantidad de aristas de un grafo $K_{n,m}$ es n * m

Ejemplos siguientes:

- Primer caso $K_{3,2}$
- Segundo caso $K_{3,3}$

In [59]:

```
plt.rcParams["figure.figsize"] = [10, 5]
plt.rcParams["figure.autolayout"] = True
fig, axes = plt.subplots(nrows=1, ncols=2)
ax = axes.flatten()

G1 = nx.Graph([('a','b'),('b','c'),('c','d'),('d','a'),('b','e'),('d','e')])
pos_dict = {'a': [-0.1, 0.9], 'b': [0.1,0.9], 'c': [-0.1,0.5], 'd': [0.1,0.5], 'e': [-0.15,0.7]}
nx.draw(G1, pos_dict, node_color="coral", font_size=10, width=1, node_size=1000, ax=ax[0])
ax[0].set_axis_off()

G2 = nx.Graph([('a','b'),('b','c'),('c','d'),('d','a'),('b','e'),('d','e'),('a','f'),('c','f'),('e','f')])
pos_dict = {'a': [-0.1, 0.9], 'b': [0.1,0.9], 'c': [-0.1,0.5], 'd': [0.1,0.5], 'e': [-0.15,0.7],'f': [0.15,0.7]}
nx.draw(G2, pos_dict, node_color="coral", font_size=10, width=1, node_size=1000, ax=ax[1])
ax[1].set_axis_off()
```



Grafos conexos

Dado un grafo G = (V, A, Φ), en el conjunto V se define la siguiente relación:

v_i R v_j ⇔ ∃ camino de v_i a v_j ∨ v_i = v_j

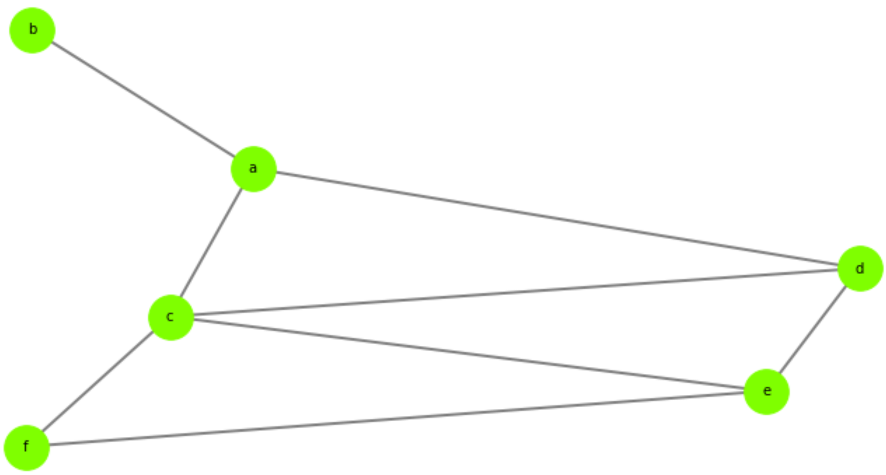
Esta relación es de equivalencia y por lo tanto pueden hallarse las clases de equivalencia, a las que se denomina componentes conexas.

Grafo conexo:

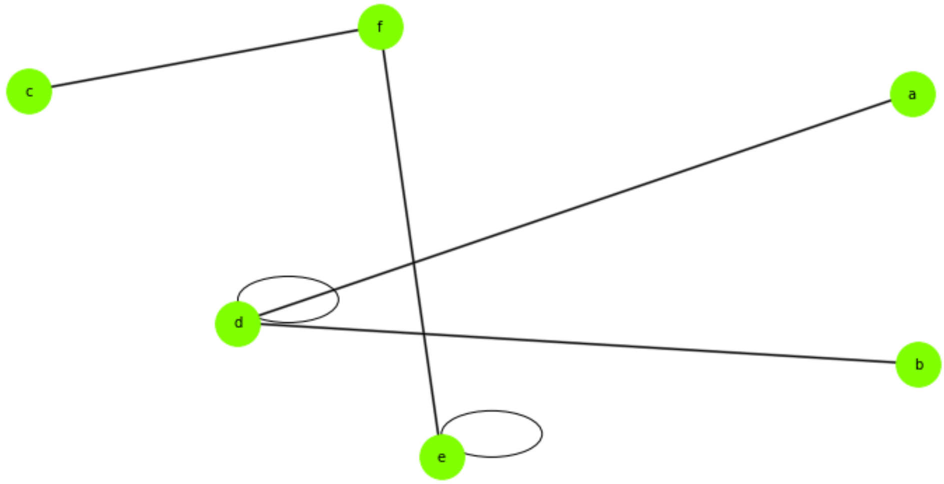
- **Un grafo es conexo si y sólo si tienen una única componente conexa.**
- **Un grafo es conexo si y sólo si existe algún camino entre todo par de vértices.**

El siguiente grafo es conexo porque de cualquier vértice se puede llegar a cualquier otro a través de un cami
no.

In [60]:
```python
L1 = ['a', 'b', 'c', 'd', 'e', 'f']
L2 = [('a', 'b'), ('a', 'c'), ('c', 'd'), ('d', 'a'),('e', 'c'), ('e', 'd'), ('e', 'f'), ('f', 'c')]
G = nx.Graph()
G.add_nodes_from(L1)
G.add_edges_from(L2)
nx.draw(G, node_color="chartreuse", edge_color="gray", font_size=10, width=2, with_labels=True, node_size=1000)
```



El siguiente grafo no es conexo porque, por ejemplo, no existe ningún camino entre los vértices a y c.



Sin embargo, está formado por dos subgrafos que cada uno de ellos sí es conexo, se llaman componentes conexa
s.

Desconexión de grafos

Dado un grafo G = (V , A , Φ):

Istmo o punto de corte: v ∈ V es istmo ⇔ ~Gv, es no conexo. O sea, un istmo es un vértice tal que al suprimirlo desconecta al grafo.

Puente: a ∈ A es puente ⇔ ~Ga, es no conexo. O sea, un puente es una arista tal que al suprimirla desconecta al grafo.

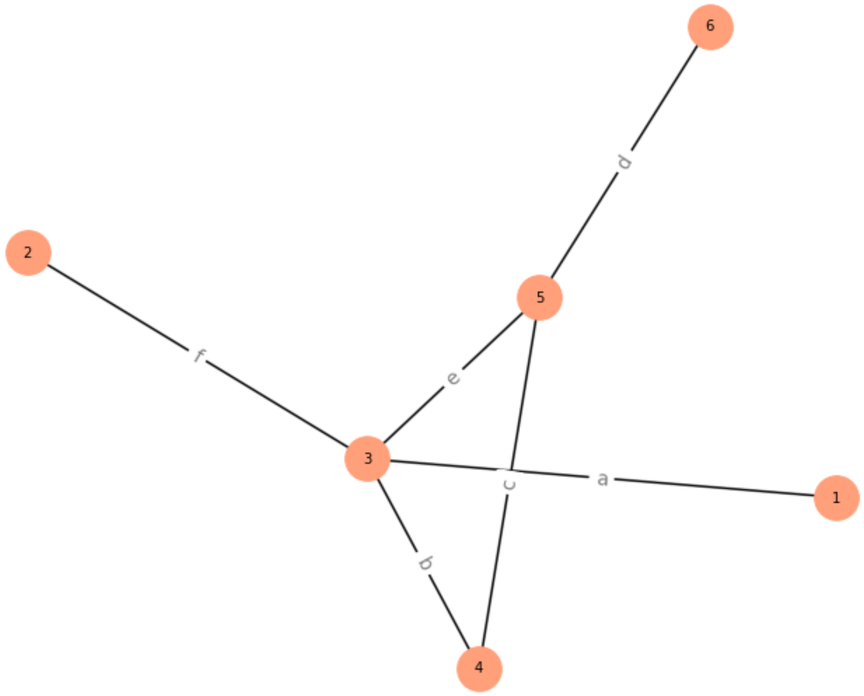
Conjunto desconectante: B ⊆ A es desconectante ⇔ ~GB, es no conexo. O sea, un conjunto de aristas es deconectante si al suprimirlas desconecta.

Conjunto de corte

B ⊆ A es de corte ⇔ B, es desconectante y además ∀ C ⊂ B, no es desconectante. O sea, un conjunto de aristas es de corte si al suprimirlo desconecta al grafo, pero ningún subconjunto propio debe hacerlo, es decir, el conjunto de corte está formado únicamente por las aristas necesarias para desconectar y no por otras.

In [61]:

```
plt.rcParams["figure.figsize"] = (10,8)
plt.rcParams["figure.autolayout"] = True
fig, ax = plt.subplots()

G = nx.Graph([(1,3,{'label':'a'}),(2,3,{'label':'f'}),(3,4,{'label':'b'}),
                (3,5, {'label':'e'}),(4,5,{'label':'c'}),(5,6,{'label': 'd'})])
pos = nx.spring_layout(G,k=2,scale=3.0)
labels = nx.get_edge_attributes(G,'label')

nx.draw(G, pos,  with_labels=True, connectionstyle="arc3,rad=0.3",
        node_color="lightsalmon", font_size=10, width=1, node_size=1000)
nx.draw_networkx_edges(G, pos, connectionstyle='arc3,rad=0.1', width = 2,alpha = 0.5)
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, font_size=14, font_color='gray')

plt.show()
```



Istmos: vértice 3 y vértice 5.

Puentes: arista a, b y d.

Conjuntos desconectantes: B_1 = {b,e}, B_2 = {a,f,e}, etc.

De los conjuntos anteriores B_1 es de corte.

Subgrafos

Dado un grafo G = (V , A , Φ): se denomina subgrafo al grafo: G' = (V' , A' , Φ/A') tal que V' ⊆ V ∧ A' ⊆ A ∧ Φ/A' es la función Φ restringida a A'

Para obtener subgrafos de un grafo dado se puede:

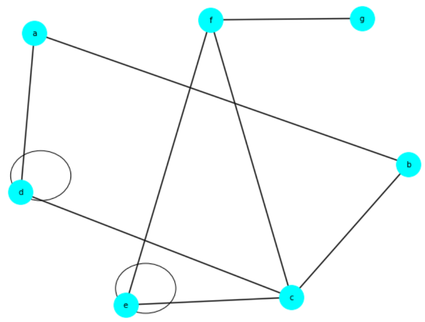
- **suprimir uno o varios vértices y las aristas incidentes en ellos.**
- **sumprimir sólo una o varias aristas.**

Si se suprime un vértice v, el subgrafo restante es ~Gv

Si se suprime un vértice a, el subgrafo restante es ~Ga

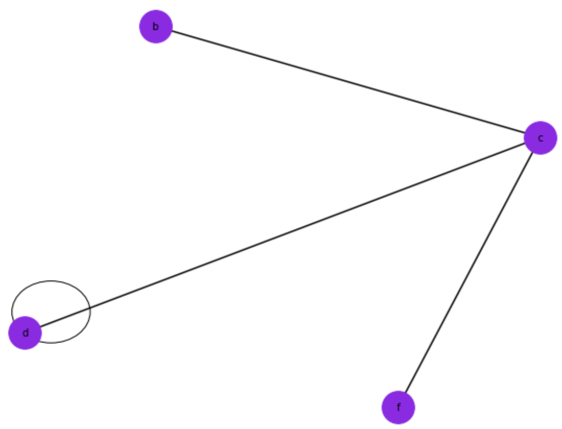
También se puede obtener un subgrafo generado por un conjunto de vértices.

Dado un grafo G = (V , A , Φ), algunos ejemplos pueden ser:

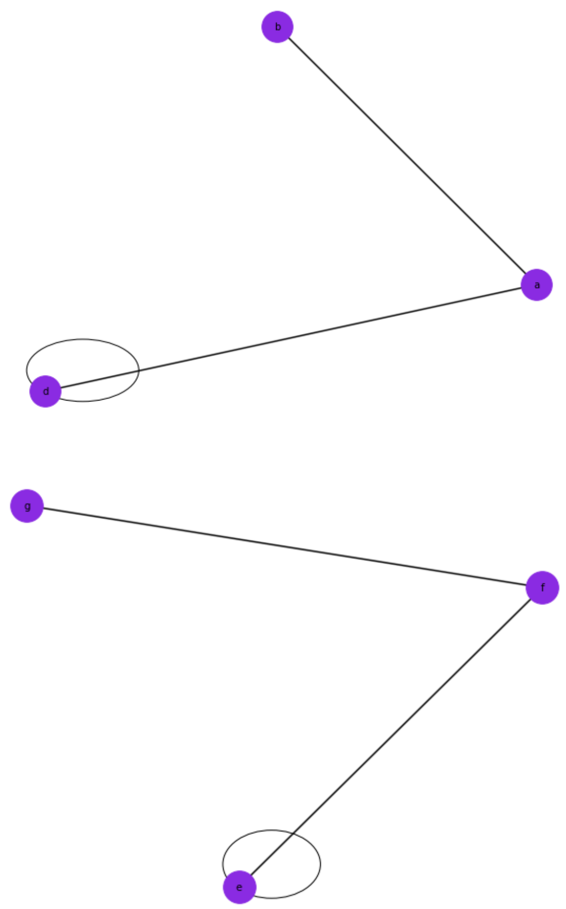



```
Algunos subgrafos son:
```

$\sim G_{a,e,g}$



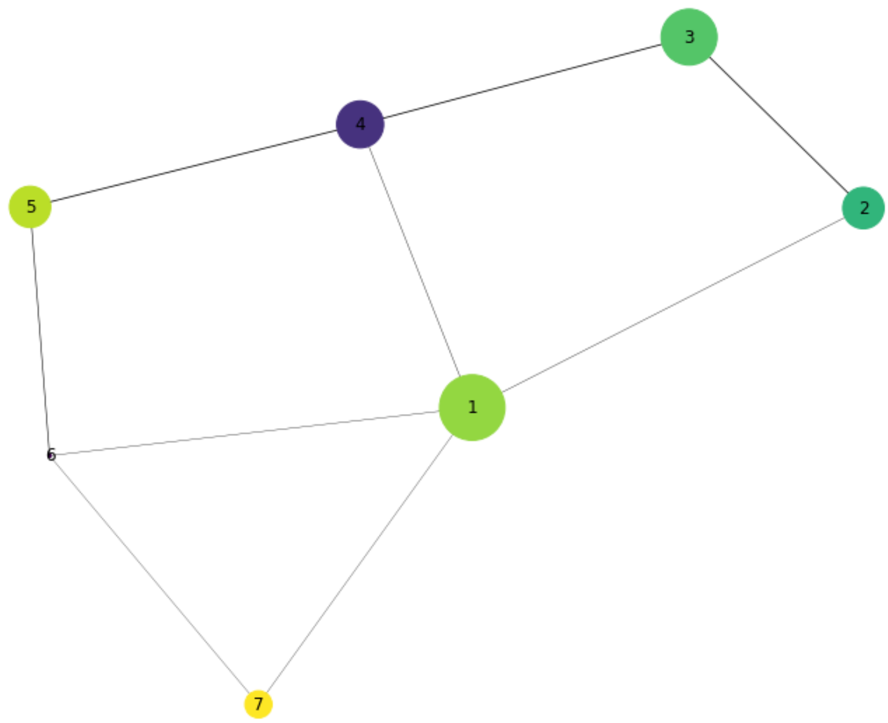
$\sim G_{c}$



Manipular atributos de nodos y bordes usando Numpy

In [62]:
```python
graph = nx.Graph()
edges = [(1,2),(2,3),(3,4),(4,5),(5,6),(6,1),(1,4),(1,7),(6,7)]
graph.add_edges_from(edges)
graph.nodes[1]
graph.nodes[1]['category'] = 'A'
print(graph.nodes[1])

graph.edges[1, 2]
graph.edges[1, 2]['weight'] = 2
print(graph.edges[1,2])

edge_weights = {edge: np.random.rand() for edge in graph.edges}
nx.set_edge_attributes(graph, edge_weights, 'weight')
graph.edges[3, 4]

node_sizes = {node: np.random.rand() * 3000 for node in graph.nodes}

nx.set_node_attributes(graph, node_sizes, 'size')
graph.nodes[5]

node_colors = {node: np.random.rand() for node in graph.nodes}

nx.set_node_attributes(graph, node_colors, 'color')
print(node_colors)

width = list(nx.get_edge_attributes(graph, 'weight').values())
node_size = list(nx.get_node_attributes(graph, 'size').values())
node_color = list(nx.get_node_attributes(graph, 'color').values())
nx.draw(graph, width = width, node_size = node_size, node_color = node_color, with_labels=True)
plt.show()
nx.get_node_attributes(graph, 'size')
```

```
{'category': 'A'}
{'weight': 2}
{1: 0.8260696227869275, 2: 0.6554653678680004, 3: 0.7258802749927181, 4: 0.17274643466276474, 5: 0.88137038928452
28, 6: 0.03837212168611959, 7: 0.9789908356820265}
```



Out[62]:
```
{1: 2187.2612386293586,
 2: 873.7441193582898,
 3: 1592.3309384881452,
 4: 1122.8418207740774,
 5: 866.195786547408,
 6: 5.259128319802464,
 7: 374.1077885904943}
```

Algunos algoritmos de aplicación

In [63]:
```
L1 = ['a', 'b', 'c', 'd', 'e', 'f']
L2 = [('a', 'b'), ('a', 'c'), ('c', 'd'), ('d', 'a'),('e', 'c'), ('e', 'd'), ('e', 'f'), ('f', 'c')]
G = nx.Graph()
G.add_nodes_from(L1)
G.add_edges_from(L2)
```

shortest_path(): *Calcula las rutas más cortas en el gráfico.*

In [64]:
```
print("Ruta mas corta entre a y e:\n ", nx.algorithms.shortest_path(G, 'a', 'e'))
```

```
Ruta mas corta entre a y e:
  ['a', 'c', 'e']
```

average_shortest_path_length(): *Devuelve la longitud promedio de ruta más corta.*

In [65]:
```
print("Promedio de la ruta mas corta:\n", nx.algorithms.average_shortest_path_length(G))
```

```
Promedio de la ruta mas corta:
 1.6
```

all_pairs_shortest_path(): *Calcula las rutas más cortas entre todos los nodos.*

In [66]:
```
print("Relación de ruta mas corta entre pares de nodos relacionado con e:\n ", dict(nx.all_pairs_shortest_path(G))
```

```
Relación de ruta mas corta entre pares de nodos relacionado con e:
  {'e': ['e'], 'c': ['e', 'c'], 'd': ['e', 'd'], 'f': ['e', 'f'], 'a': ['e', 'c', 'a'], 'b': ['e', 'c', 'a',
'b']}
```

dijkstra_path()

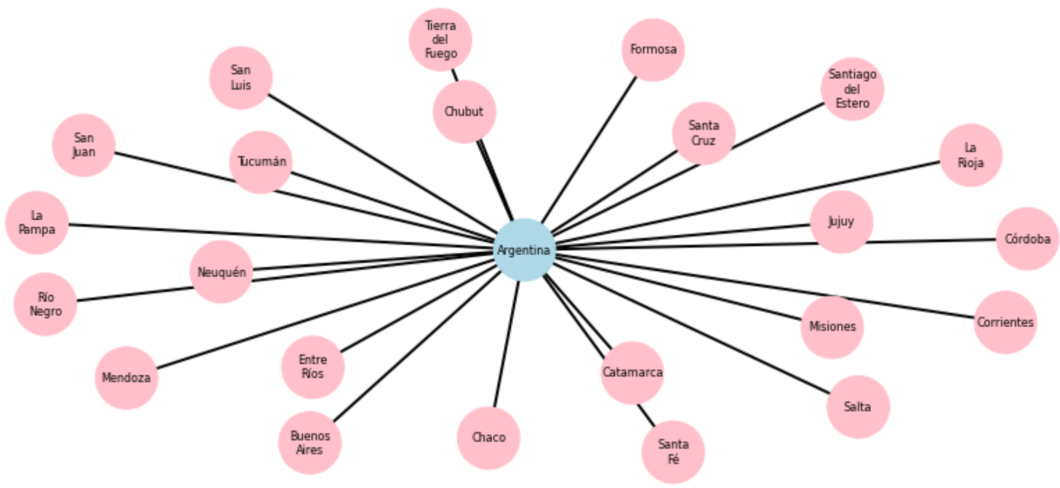
In [67]:
```
print("Ruta mas corta usando el algoritmo de Dijkstra entre a y e:\n", nx.algorithms.dijkstra_path(G,'a','e'))
```

```
Ruta mas corta usando el algoritmo de Dijkstra entre a y e:
 ['a', 'c', 'e']
```

Ejemplos de aplicación y funciones

In [68]:
```python
plt.rcParams['figure.figsize'] = (12.0, 5.0)
G = nx.Graph()
G.add_nodes_from((["Argentina", "Jujuy", "Salta", "Tucumán", "Catamarca", "Santiago\ndel\nEstero",
                   "Formosa", "Chaco", "Misiones", "Corrientes",
                   "Entre\nRíos", "Santa\nFé", "Córdoba", "Buenos\nAires", "La\nPampa",
                   "San\nJuan", "San\nLuis", "Mendoza", "La\nRioja",
                   "Neuquén", "Río\nNegro", "Chubut", "Santa\nCruz", "Tierra\ndel\nFuego"]))
G.add_edges_from([("Jujuy", "Argentina"),
                  ("Salta", "Argentina"),
                  ("Tucumán",  "Argentina"),
                  ("Catamarca", "Argentina"),
                  ("Santiago\ndel\nEstero", "Argentina"),
                  ("Formosa", "Argentina"),
                  ("Chaco", "Argentina"),
                  ("Misiones", "Argentina"),
                  ("Corrientes", "Argentina"),
                  ("Entre\nRíos", "Argentina"),
                  ("Santa\nFé", "Argentina"),
                  ("Córdoba", "Argentina"),
                  ("Buenos\nAires", "Argentina"),
                  ("La\nPampa", "Argentina"),
                  ("San\nJuan", "Argentina"),
                  ("San\nLuis", "Argentina"),
                  ("Mendoza", "Argentina"),
                  ("La\nRioja", "Argentina"),
                  ("Neuquén", "Argentina"),
                  ("Río\nNegro", "Argentina"),
                  ("Chubut", "Argentina"),
                  ("Santa\nCruz", "Argentina"),
                  ("Tierra\ndel\nFuego", "Argentina")])
pais = ["Argentina"]
nx.draw(G, node_color=['lightblue' if node in pais else 'pink' for node in G.nodes()],
        edge_color="black", font_size=8, width=2, with_labels=True, node_size=2000)
plt.figure(figsize=(12,10))
```

Out[68]: <Figure size 864x720 with 0 Axes>



<Figure size 864x720 with 0 Axes>

radius()

In [69]:
```python
print("Radio: %d\n" % nx.radius(G))
```

Radio: 1

diameter()

In [70]:
```python
print("Diámetro: %d\n" % nx.diameter(G))
```

Diámetro: 2

eccentricity()

In [71]:
```
print("Excentricidad: %s\n" % nx.eccentricity(G))
```

```
Excentricidad: {'Argentina': 1, 'Jujuy': 2, 'Salta': 2, 'Tucumán': 2, 'Catamarca': 2, 'Santiago\ndel\nEstero': 2,
'Formosa': 2, 'Chaco': 2, 'Misiones': 2, 'Corrientes': 2, 'Entre\nRíos': 2, 'Santa\nFé': 2, 'Córdoba': 2, 'Buenos
\nAires': 2, 'La\nPampa': 2, 'San\nJuan': 2, 'San\nLuis': 2, 'Mendoza': 2, 'La\nRioja': 2, 'Neuquén': 2, 'Río\nNe
gro': 2, 'Chubut': 2, 'Santa\nCruz': 2, 'Tierra\ndel\nFuego': 2}
```

center()

In [72]:
```
print("Centro: %s\n" % nx.center(G))
```

```
Centro: ['Argentina']
```

periphery()

In [73]:
```
print("Periferia: %s\n" % nx.periphery(G))
```

```
Periferia: ['Jujuy', 'Salta', 'Tucumán', 'Catamarca', 'Santiago\ndel\nEstero', 'Formosa', 'Chaco', 'Misiones', 'C
orrientes', 'Entre\nRíos', 'Santa\nFé', 'Córdoba', 'Buenos\nAires', 'La\nPampa', 'San\nJuan', 'San\nLuis', 'Mendo
za', 'La\nRioja', 'Neuquén', 'Río\nNegro', 'Chubut', 'Santa\nCruz', 'Tierra\ndel\nFuego']
```

density()

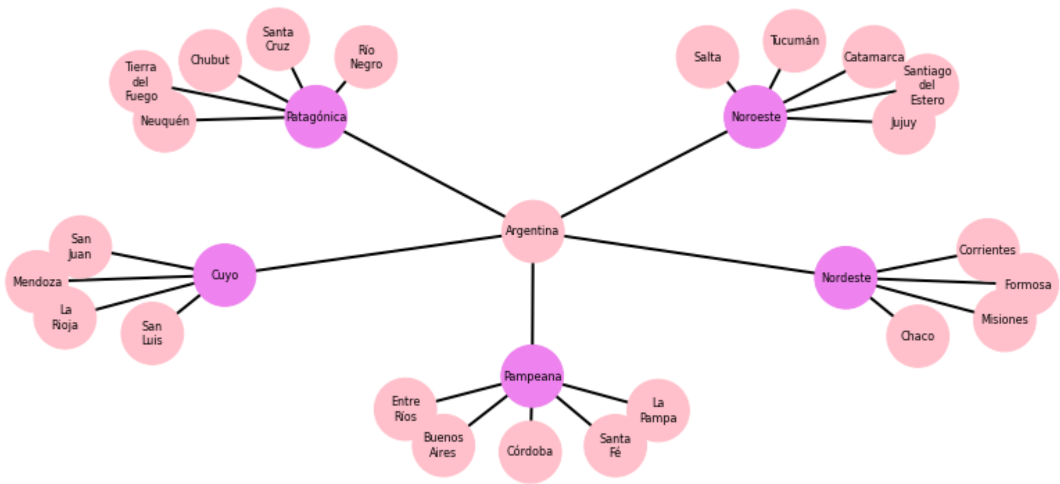
In [74]:
```
print("Densidad: %s\n" % nx.density(G))
```

```
Densidad: 0.08333333333333333
```

In [75]:

```python
plt.rcParams['figure.figsize'] = (12.0, 5.0)
G = nx.Graph()
G.add_node("Argentina")
G.add_nodes_from(["Noroeste", "Nordeste", "Pampeana",
                  "Cuyo", "Patagónica"])
G.add_nodes_from(("Jujuy", "Salta", "Tucumán", "Catamarca", "Santiago\ndel\nEstero",
                   "Formosa", "Chaco", "Misiones", "Corrientes",
                   "Entre\nRíos", "Santa\nFé", "Córdoba", "Buenos\nAires", "La\nPampa",
                   "San\nJuan", "San\nLuis", "Mendoza", "La\nRioja",
                   "Neuquén", "Río\nNegro", "Chubut", "Santa\nCruz", "Tierra\ndel\nFuego"]))
G.add_edge("Noroeste", "Argentina")
G.add_edge("Nordeste", "Argentina")
G.add_edge("Pampeana", "Argentina")
G.add_edge("Cuyo", "Argentina")
G.add_edge("Patagónica", "Argentina")
lista_noroeste = [("Jujuy",  "Noroeste"), ("Salta", "Noroeste"),
                  ("Catamarca", "Noroeste"), ("Santiago\ndel\nEstero", "Noroeste"), ("Tucumán", "Noroeste")]
G.add_edges_from(lista_noroeste)
lista_nordeste = [("Formosa", "Nordeste"), ("Chaco", "Nordeste"),
                  ("Misiones", "Nordeste"), ("Corrientes", "Nordeste")]
G.add_edges_from(lista_nordeste)
lista_pampeana = [("Entre\nRíos", "Pampeana"), ("Santa\nFé", "Pampeana"),
                  ("Córdoba", "Pampeana"),("La\nPampa", "Pampeana"), ("Buenos\nAires", "Pampeana")]
G.add_edges_from(lista_pampeana)
lista_cuyo = [("San\nJuan", "Cuyo"), ("San\nLuis", "Cuyo"),
              ("Mendoza", "Cuyo"), ("La\nRioja", "Cuyo")]
G.add_edges_from(lista_cuyo)
lista_patagonica = [("Neuquén", "Patagónica"), ("Río\nNegro", "Patagónica"), ("Chubut", "Patagónica"),
                    ("Santa\nCruz", "Patagónica"), ("Tierra\ndel\nFuego", "Patagónica")]
G.add_edges_from(lista_patagonica)
regiones = ["Noroeste", "Nordeste", "Pampeana","Cuyo", "Patagónica"]
nx.draw(G, node_color= ['pink' if not node in regiones else 'violet' for node in G.nodes()],
        edge_color="black", font_size=8, width=2, with_labels=True, node_size=2000)
plt.figure(figsize=(12,10))
plt.axis('off')
```

Out[75]: (0.0, 1.0, 0.0, 1.0)



In [76]:
```
plt.rcParams['figure.figsize'] = (15.0, 7.5)
aeropuertos = pd.read_csv('archs/15.aeropuertos.csv', encoding='latin-1')
df_a = pd.DataFrame(aeropuertos)
df_a.head(5)
```

Out[76]:

	COD	CIUDAD	AEROPUERTO	PROVINCIA
0	AEP	Ciudad de Buenos Aires	Jorge Newbery	Buenos Aires
1	EZE	Ezeiza	Ministro Pistarini	Buenos Aires
2	JNI	Junín	Junín	Buenos Aires
3	LPG	La Plata	La Plata	Buenos Aires
4	MDQ	Mar del Plata	Astor Piazzolla	Buenos Aires

In [77]:
```
vuelos = pd.read_csv("archs/15.combi_precios.csv", encoding='latin-1')
df_b = pd.DataFrame(vuelos)
df_b.head(5)
```

Out[77]:

	Origen	Destino	Duracion	Precio
0	AEP	CNQ	95.45	680.0
1	EZE	IRJ	39.50	4780.0
2	JNI	COC	51.44	1160.0
3	LPG	AEP	66.26	7580.0
4	MDQ	GPO	18.85	720.0

In [78]:
```
df_b.info()
```

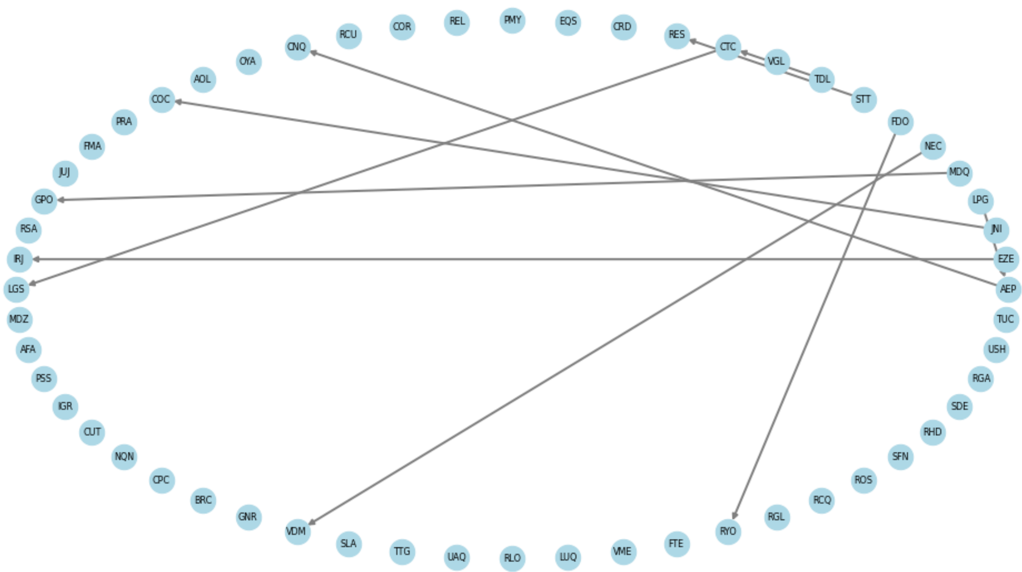
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 176 entries, 0 to 175
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   Origen    176 non-null    object 
 1   Destino   176 non-null    object 
 2   Duracion  176 non-null    float64
 3   Precio    176 non-null    float64
dtypes: float64(2), object(2)
memory usage: 5.6+ KB
```

In [79]:
```python
df_p = df_b.iloc[0:10,0:4]
```

In [80]:
```python
DG = nx.DiGraph()

for i in range(0, len(df_a)):
    DG.add_node(df_a.iloc[i]['COD'])
    i = i + 1
```

In [81]:
```python
for i in range(0, len(df_p)):
    DG.add_edge(df_p.iloc[i]['Origen'], df_p.iloc[i]['Destino'])
    i = i + 1
```

In [82]:
```python
DG.nodes(data=True)
nx.draw_circular(DG,
                 node_color="lightblue",
                 edge_color="gray",
                 font_size=8,
                 width=2, with_labels=True, node_size=500,
                 )
plt.figure(figsize=(10,7))
plt.show()
```




```
<Figure size 720x504 with 0 Axes>
```

In [83]:
```python
df_b['weight'] = round(df_b['Precio'] / df_b['Duracion'],2)
df_b.head()
```

Out[83]:

	Origen	Destino	Duracion	Precio	weight
0	AEP	CNQ	95.45	680.0	7.12
1	EZE	IRJ	39.50	4780.0	121.01
2	JNI	COC	51.44	1160.0	22.55
3	LPG	AEP	66.26	7580.0	114.40
4	MDQ	GPO	18.85	720.0	38.20

In [84]:
```python
df_b = df_b.loc[0:15, ['Origen', 'Destino', 'weight']]
```

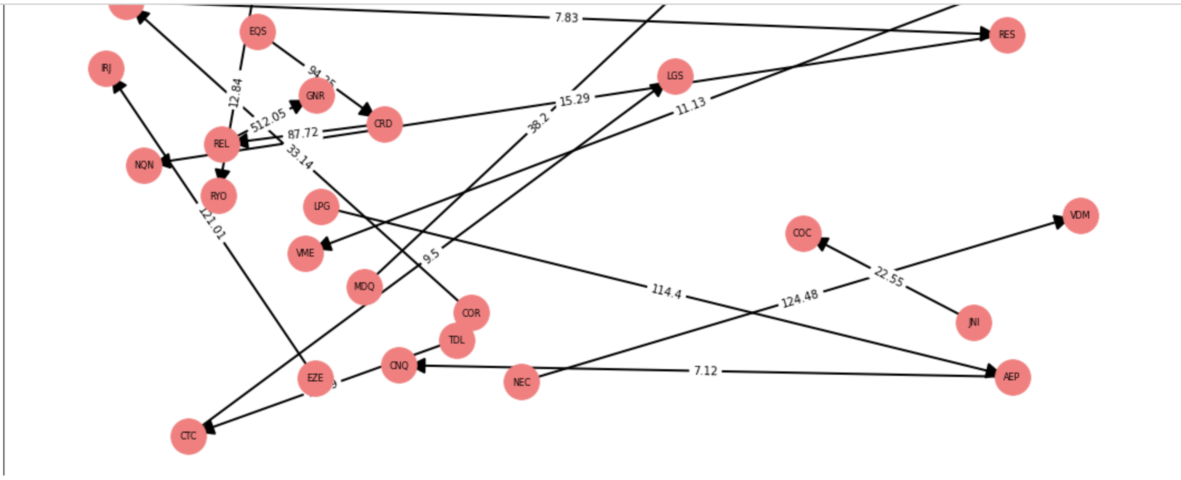
In [85]:
```python
DG = nx.DiGraph()

DG.add_weighted_edges_from([tuple(x) for x in df_b.values])
DG.edges()
```

Out[85]:
```
OutEdgeView([('AEP', 'CNQ'), ('EZE', 'IRJ'), ('JNI', 'COC'), ('LPG', 'AEP'), ('MDQ', 'GPO'), ('NEC', 'VDM'), ('FD
O', 'RYO'), ('STT', 'RES'), ('RES', 'NQN'), ('TDL', 'CTC'), ('CTC', 'LGS'), ('CRD', 'REL'), ('REL', 'GNR'), ('EQ
S', 'CRD'), ('PMY', 'VME'), ('COR', 'STT')])
```

In [86]:
```python
DG.get_edge_data('AEP','CNQ')
```

Out[86]:
```
{'weight': 7.12}
```

In [87]:
```
pos = nx.random_layout(DG)
nx.draw_networkx_nodes(DG, pos, node_color="lightcoral", node_size=1000)
nx.draw_networkx_labels(DG, pos, font_size=8, font_family='sans-serif')
labels = nx.get_edge_attributes(DG, 'weight')
nx.draw_networkx_edges(DG, pos, width=2, arrowstyle= '-|>', arrowsize = 30)
nx.draw_networkx_edge_labels(DG, pos, edge_labels=labels)

plt.figure(figsize=(7,7))
plt.axis('off')
plt.show()
```



¿Qué clase de grafo necesito representar?

Clase Networkx	Tipo	Autoloops permitidos	Bordes paralelos permitidos	Documentación
Graph	no dirigido	si	No	https://networkx.org/documentation/stable/reference/classes/graph.html (https://networkx.org/documentation/stable/reference/classes/graph.html)
DiGraph	dirigido	si	No	https://networkx.org/documentation/stable/reference/classes/digraph.html (https://networkx.org/documentation/stable/reference/classes/digraph.html)
MultiGraph	no dirigido	si	si	https://networkx.org/documentation/stable/reference/classes/multigraph.html (https://networkx.org/documentation/stable/reference/classes/multigraph.html)
MultiDiGraph	dirigido	si	si	https://networkx.org/documentation/stable/reference/classes/multidigraph.html (https://networkx.org/documentation/stable/reference/classes/multidigraph.html)

Más Información (https://networkx.guide/visualization/)

Más Información (https://networkx.org/documentation/stable/auto_examples/basic/plot_simple_graph.html)