# NETWORKX

Networkx היא ספריית קוד פתוח בשפת פייתון ליצירת, לימוד וביצוע מניפולציות על גרפים.
הספרייה מספקת:
- כלים לחקר מבנה והדינמיקה של רשתות חברתיות, ביולוגיות ותשתיתיות.
- ממשק תכנותי סטנדרטי ויישום של גרפים שמתאימים לאפליקציות רבות.
- היכולת לעבודה במאמץ מינימאלי עם דאטה סטים לא סטנדרטים.

היות והספרייה ענקית נמליץ מראש להסתכל על האתר הרשמי של הספרייה לעוד פרטים.

**התקנה:**

```
pip install networkx
```

## פעולות בסיסיות בגרפים-

networx מספקת כלי לשימוש בdata farme לא סטנדרטי כל כך- בפנדס היה לנו מבנה רלאציוני שנשען על האובייקט מערך של numpy, networkx לעומת זאת משתמשת במבנה של גרף לשמירה על הנתונים, וכגרף ניתן לבצע עליו מניפולציות מתוך תורת הגרפים למשל מציאת שידוך מקסימלי, מציאת מסלול קצר, קוטר צביעה ועוד.

תחילה נראה כיצד לייצר גרף, להוסיף לו קשתות וצמתים.
יצירת גרף לא מכוון תעשה דרך שימוש במחלקה Graph של הספרייה, או במחלקה DiGraph ליצירת גרף מכוון, אפשר לאתחל את הגרף מראש עם גרף אחר, או עם רשימה של צלעות.
בשביל להוסיף צמתים לגרף נשתמש בכמה פונקציות, הראשונה היא המתודה add_node() שמקבלת את שם הצומת ושומרת אותו כקודקוד בודד.
הדרך השנייה היא להוסיף ישירות שני קודקודים, עם המתודה add_nodes_from() המתודה מקבלת רשימה של כמה קודקודים ומכניסה אותם לגרף, ואפשר גם להוסיף תכונות לקודקוד אם מוסיפים לו בנוסף לערך גם מילון עם התכונות של אותו קודקוד:

```
import networkx as nx

my_graph = nx.Graph()
my_graph.add_node(1)
my_graph.add_nodes_from([2, 3])
```

אפשר גם ליצור גרף רנדומלי עם הפונקציה path_graph() ולספק לה מספר ממנו היא תיקח את ערכיה.
וכמו כן אפשר גם להכניס גרף אחד באחר(להכניס את הצמתים שלו אחד אחד) עם המתודה add_node_from() , או להכניס גרף אחר כצומת אחד בגרף עם מתודה add_node(), למה שנרצה לעשות את זה? תחשבו על האינטרנט, היא רשת של רשתות, יש לה רשת ברמת המקרו ורשת פנימית ברמת המיקרו:

```
graph_other = nx.path_graph(10)
my_graph.add_nodes_from(graph_other)
my_graph.add_node(graph_other)
```

הוספת ענפים בגרף נעשה עם המתודה add_edge() ונכניס בה את הצמתים שבניהם קיימת קשת.
אפשר גם להכניס כמה קשתות כאשר כל קשת מצוינת ב-tuple עם ערכי שני קוקודים והתכונה שאותה נושאת הקשת(למשל משקל) במילון. כל גרף שומר את הקשתות שלו במשתנה בשם edges:

```
my_graph.add_edge(1, 2)
e = (2, 3)
my_graph.add_edge(*e)

my_graph.add_edges_from([(1, 2), (1, 3),(2, 3, {'weight': 3.1415})])

list(my_graph.edges)
```

```
[(1, 2), (1, 3), (2, 3)]
```

```
my_graph.clear()
```

בשביל לנקות את הגרף נשתמש בפונקציה clear().

לכל הגרפים יש כמה שדות: צמתים, קשתות, טבלאת שכנים, טבלת דרגות בגרף.
יש גם את הפונקצייה successor שמאפשרת לראות אילו קודקודים מחוברים לקודקוד ספציפי בקשת.

```
G = nx.DiGraph()
G.add_edge(2, 1)   # adds the nodes in order 2, 1
G.add_edge(1, 3)
G.add_edge(2, 4)
G.add_edge(1, 2)
G.add_edge(3, 2)
assert list(G.successors(2)) == [1, 4]
assert list(G.edges) == [(2, 1), (2, 4), (1, 3), (1, 2),(3, 2)]
print(list(G.nodes))
print(list(G.edges))
print(list(G.adj[1]))  # or list(G.neighbors(1))
print(G.degree[1])  # the number of edges incident to 1
```

```
[2, 1, 3, 4]
[(2, 1), (2, 4), (1, 3), (1, 2), (3, 2)]
[3, 2]
3
```

הסרה של צמתים או קשתות תעשה עם המתודות remove_node() להוצאת קדקוד בודד כשהארגומנט הוא שם הקדקוד, והוצאת כל הקדקודים שמחוברים לקדקוד ספציפי עם המתודה remove_nodes_from().
הוצאת צלע ספציפית עם המתודה remove_edge() והארגומנט צריך להיות הקדקודים שיש בניהם צלע.
והוצאת כמה צלעות עם המתודה remove_edges_from() והארגומנט שלה הוא רשימה של צלעות שצריך להוציא מהגרף.

```
G.add_edge(4,1)
G.add_edge(4,5)
G.add_edge(5,1)
G.add_edge(2,5)
G.add_edge(6,5)
G.add_edge(1,6)
G.add_edge(3,5)
G.add_edge(4,6)
G.remove_node(3)
G.remove_nodes_from('4')
G.remove_nodes_from('5')
G.remove_edge(6, 5)
```

כל אובייקט בגרף (צומת או צלע) יכול להכיל נתונים לגביו שנשמרים במילון כפי שראינו קודם.
אפשר לגשת ישירות לשכנים של צומת ספציפי, ללא עזרת adj[] , עם הפרמטר [], ובשביל לגשת ישירות לצלע נכניס את שני הקדקודים לתוך הפרמטר כמו במערך דו ממדי. זה יספק לנו מידע על הצלע ועל התכונות שלה.

בנוסף אפשר לגשת ישירות לתכונה של צלע אם נוסיף לשני האופרטורים לעיל עוד אופרטור [] שישמש עבור המילון של התכונות. משום שזה מילון ניתן גם להוסיף תכונה חדשה מבלי להשתמש בפונקציה ייעודית לכך:

```
G = nx.Graph([(1, 2, {"color": "yellow"})])
print(G[1])  # same as G.adj[1]
print(G[1][2])
print(G[1][2]['color'])
G.add_edge(1, 3)
G[1][3]['color']= 'green'
print(G[1][3]['color'])
```

```
{2: {'color': 'yellow'}}
{'color': 'yellow'}
yellow
green
```

באופן דומה אפשר לאתחל גרף עם תכונות, למשל שם הגרף, תפקידו וכו', או להוסיף לו תכונות יותר מאוחר עם המשתנה graph של הגרף, שהוא מילון התכונות שלו:

```
G = nx.Graph(day="Friday")
print(G.graph)
G.graph['year'] = "2021"
print(G.graph)
```

```
{'day': 'Friday'}
{'day': 'Friday', 'year': '2021'}
```

אם נרצה לבחון מהר את כל הצמתים ושכניהם בארכיון נוכל להשתמש במתודות adjacency() או במתודה item() של המשתנה adj . יש לשים לב שבגרף לא מכוון תוצג כל צלע פעמיים:

```
FG = nx.Graph()
FG.add_weighted_edges_from([(1, 2, 0.125), (1, 3, 0.75), (2, 4, 1.2), (3, 4, 0.375)])
for n, nbrs in FG.adj.items():
   for nbr, eattr in nbrs.items():
       wt = eattr['weight']
       if wt < 0.5: print(f"({n}, {nbr}, {wt:.3})")
print()
for (u, v, wt) in FG.edges.data('weight'):
    if wt < 0.5:
        print(f"({u}, {v}, {wt:.3})")
```

```
(1, 2, 0.125)
(2, 1, 0.125)
(3, 4, 0.375)
(4, 3, 0.375)

(1, 2, 0.125)
(3, 4, 0.375)
```

## יצירת גרף חדש בצורה אקראית או אופרטיבית-

חוץ מבנייה של גרף צלע אחר צלע, קדקוד אחר קדקוד, אפשר גם ליצור גרף כתוצאה של אופרציה בין שני גרפים, למשל חיבור של שני גרפים יחד, חיסור, שינוי מגרף לא מכוון לגרף מכוון ועוד:

אופרציות על גרפים:

| | |
|---|---|
| מחזיר את תת גרף שהוא חיתוך בין הגרף לקבוצת קודקודים | `subgraph(G, nbunch)` |
| מחזיר איחוד בין שני גרפים עם חזרות | `union(G, H[, rename, name])` |
| מחזיר איחוד בין שני גרפים ללא חזרות | `disjoint_union(G, H)` |

| | |
|---|---|
| מחזיר מכפלה קרטזיט בין שני גרפים | `cartesian_product(G, H)` |
| מחזיר גרף המשלב את שני הגרפים | `compose(G, H)` |
| מחזיר את הגרף המשלים לגרף שהתקבל כארגומנט | `complement(G)` |
| מחזיר העתק של הגרף, אבל ללא צלעות | `create_empty_copy(G[, with_data])` |
| מחזיר גרף לא מכוון קפוא (graph view) של הפרמטר | `to_undirected(graph)` |
| מחזיר גרף מכוון קפוא (graph view) של הפרמטר | `to_directed(graph)` |

הספרייה גם מספקת פונקציות ליצירת גרפים מוכרים, להלן כמה דוגמאות לגרפים כאלו:

| | |
|---|---|
| גרף שלם (קליק) | `complete_graph(num_of_nodes)` |
| גרף דו צדדי שלם | `complete_bipartite_graph(set_size,set_size_sec)` |
| גרף בארבל (שני גרפים שלמים מחוברים במסלול) | `barbell_graph(first_k, second_k)` |
| גרף סוכריה (גרף שלם שמחובר למסלול) | `lollipop_graph(k, path)` |

```
K_5 = nx.complete_graph(5)                  # Returns the complete graph K_n with n nodes.
K_3_5 = nx.complete_bipartite_graph(3, 5)   # Returns the complete bipartite graph K_{n_1,n_2}.
barbell = nx.barbell_graph(10, 10)          # Returns the Barbell Graph: two complete graphs connected by
                                            # a path.
lollipop = nx.lollipop_graph(10, 20)        # Returns the Lollipop Graph; K_m connected to P_n.
```

גרפים מיוחדים:

```
er = nx.erdos_renyi_graph(100, 0.15)        # Returns a Gn,p random graph, also known as an Erdős-Rényi
                                            # graph or a binomial graph.

ws = nx.watts_strogatz_graph(30, 3, 0.1)    # Returns a Watts-Strogatz small-world graph.

ba = nx.barabasi_albert_graph(100, 5)       # Returns a random graph according to the Barabási-Albert
                                            # preferential  attachment model.

red = nx.random_lobster(100, 0.9, 0.9)      # Returns a random lobster graph.
```

## *ציור בסיסי של גרפים:*

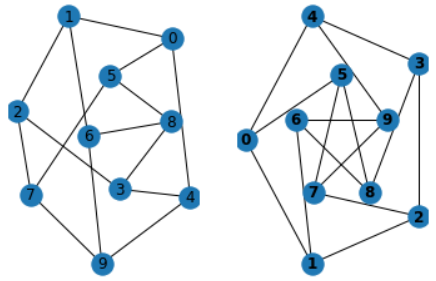
הספרייה בעיקרון לא נועדה לציור הגרפים, אבל נוכל להשתמש בספריה matplotlib כממשק עבור התת ספריה drawing של networkx .
לספרייה יש כמה פונקציות לציור גרף בסיסי, הפונקציות הבסיסיות ביותר הן draw() ו-draw_network() , והן מקבלות גרף כפרמטר אופציות להוספה לגרף למשל האם להראות תגיות על הצמתים, גדלים של הפונט או הצמתים וכו', האופציות מגיעות כ- kwargs:

```
options = {
    "font_size": 10,
    "node_size": 300,
    "node_color": "white",
    "edgecolors": "black",
    "linewidths": 1,
    "width": 1,
    "with_labels": True
}
import matplotlib.pyplot as plt
```

```
nx.draw(G ,**options)
plt.show()
```

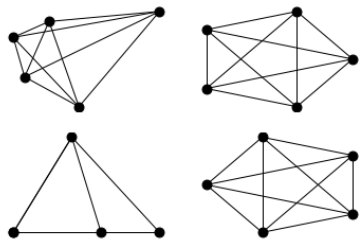
יש עוד כמה פונקציות שמאפשרות לצייר את הגרף בצורה מסוימת, למשל בתוך מעטפת כמו הפונקציה draw_shell():

```
pet = nx.petersen_graph()
plt.subplot(121)
nx.draw(pet, with_labels=True)
plt.subplot(122)
nx.draw_shell(pet, nlist=[range(5, 10), range(5)], with_labels=True, font_weight='bold')
```




או בצורות אחרות גם כן:

```
options = {
    'node_color': 'black',
    'node_size': 80,
    'width': 1,
}
plt.subplot(221)
nx.draw_random(K_5, **options)
plt.subplot(222)
nx.draw_circular(K_5, **options)
plt.subplot(223)
nx.draw_spectral(K_5, **options)
plt.subplot(224)
nx.draw_shell(K_5, **options)
```



## *פונקציות על גרפים-*

networkX מספקת מאגר עצום של פונקציות מתורת הגרפים, בניהן מציאת המסלול הקצר ביותר, מציאת שידוך בגרף, צביעה של הגרף ועוד.

דוגמא לפונקציות למציאת המסלול הקצר ביותר:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# shortest paths and path lengths between nodes in the graph.
# These algorithms work with undirected and directed graphs
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
nx.shortest_path(G)                                    # returns dictionary of shortest paths
nx.has_path(G, source, target)                         # boolean
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Shortest path algorithms for weighed graphs.
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#dijkstra
nx.dijkstra_predecessor_and_distance(G, source)        # returns dictionary of shortest path
nx.dijkstra_path(G, source, target )                   # returns list
# Floyd Warshall
nx.floyd_warshall_numpy(G)                             # returns a numpay array
```

מציא שידוך בגרף:

```
nx.max_weight_matching(G)
```

מציאת מעגל:

```
nx.find_cycle(G)
```