② グラフの実現方法

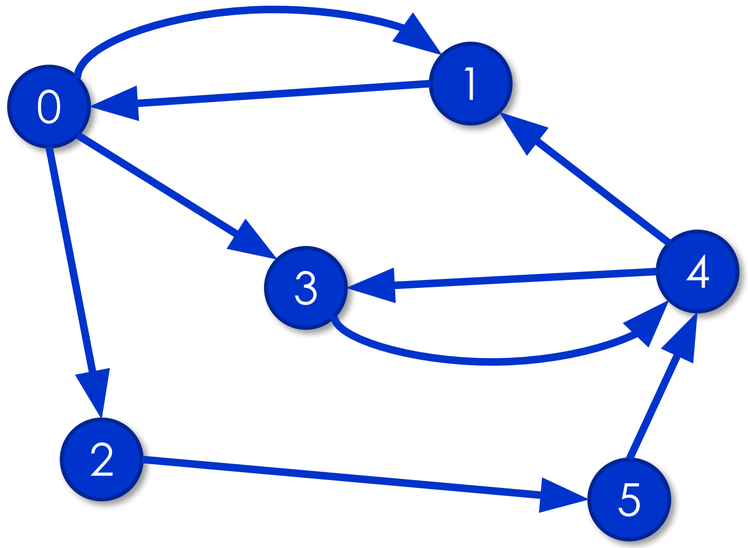
ここでは、Pythonでグラフを実装する方法を学習します。

グラフ構造の実装方法のアプローチ

- グラフ構造をプログラムで表現する際には、一般的に次の2種類のアプローチがあります

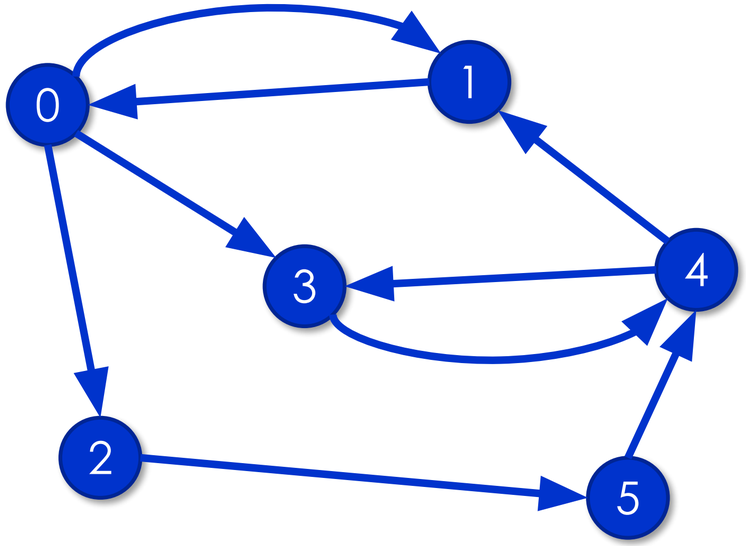
1. 隣接行列による実装
   - 2つのノード間の隣接関係を、行列（リストのリスト）で表現する
2. 隣接リストによる実装
   - ノードに隣接するノードをリストで表現する

- ここでは、それぞれの方法で、右の有向グラフを表現してみます



隣接行列

- 隣接行列の場合は、グラフのノード数が n の場合には、n x nの行列を用意します
- ノード j がノード i に隣接している場合には、 i 行目の j 列目を 1にします



	0	1	2	3	4	5
0	0	1	1	1	0	0
1	1	0	0	0	0	0
2	0	0	0	0	0	1
3	0	0	0	0	1	0
4	0	1	0	1	0	0
5	0	0	0	0	1	0

Pythonで表現すると…

- Pythonでは、以下のように「リストのリスト」として、簡単に表現できますね

```
graph1 = [
    [0, 1, 1, 1, 0, 0],
    [1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 1, 0],
    [0, 1, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0]]
```

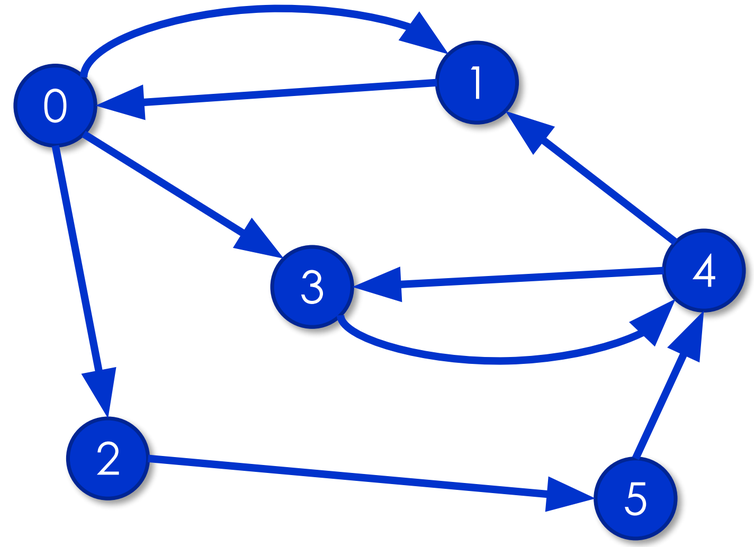
右のグラフを表現しています

```
def adjacent_matrix(matrix, i, j):
    return matrix[i][j] == 1
```

i番目とj番目の隣接をチェックします

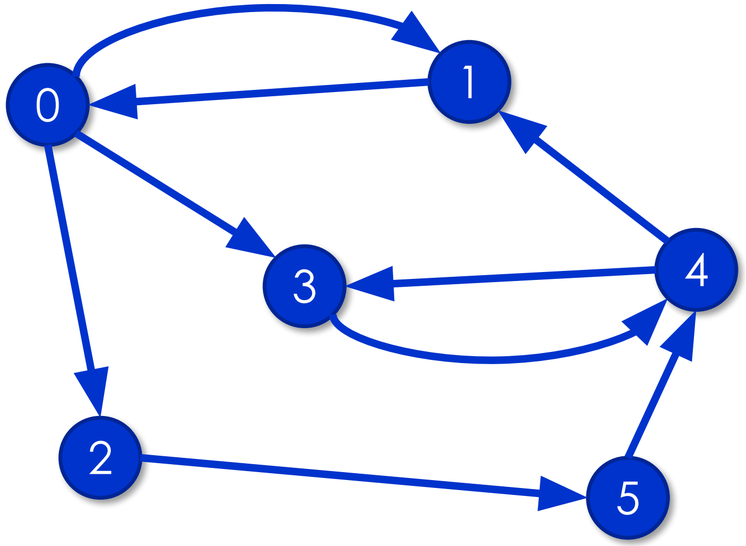
```
adjacent_matrix(graph1, 0, 2)
```

```
True
```



隣接リスト

- 隣接リストの場合は、グラフのノード数が n の場合には、ノードに対応したn個のリストを用意します
- ノード j がノード i に隣接している場合には、ノード i に対応したリストに、 ノード j のインデックスやリンクを追加します



0	1	2	3
1	0		
2	5		
3	4		
4	1	3	
5	4		

Pythonで表現すると…

- Pythonでは、以下のように表現できます
  - リンクリストを用いることも多くありますが、ここでは通常のリストとしています

```
graph2 = [
    [1, 2, 3],
    [0],
    [5],
    [4],
    [1, 3],
    [4]]
```

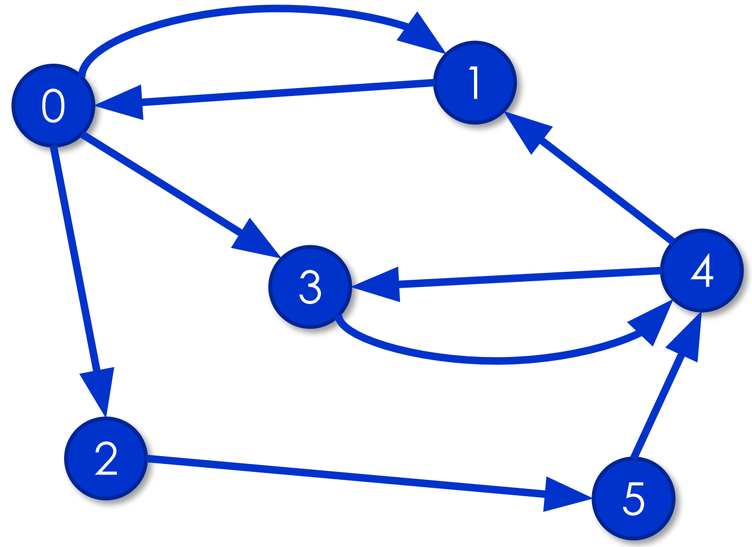
右のグラフを表現しています

```
def adjacent_list(lst, i, j):
    return j in lst[i]
```

i番目とj番目の隣接をチェックします

```
adjacent_list(graph2, 0, 2)
```

```
True
```



隣接行列 vs 隣接リスト

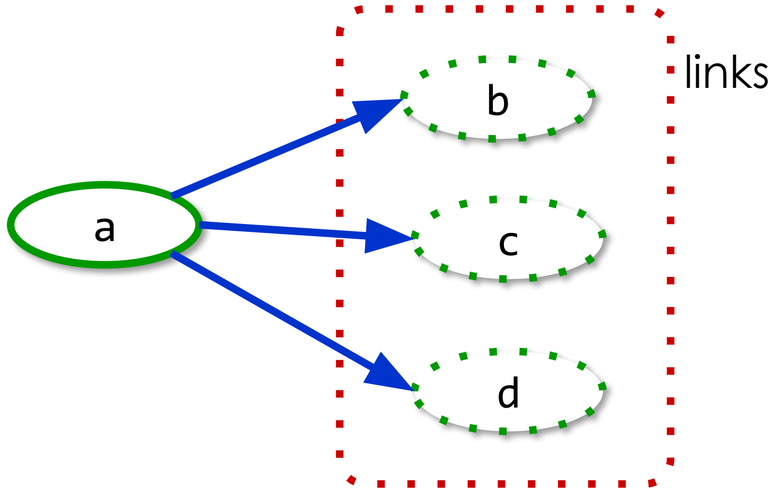
- 隣接行列と隣接リストは一長一短があります
- 隣接行列
  - 2つのノードの隣接をすぐに判断することができる
  - あるノードに隣接するノードを探索するには、全ノードをチェックする必要がある
  - 密なグラフ（ノードに対して多くのリンクがある）を効率的に表現できる
- 隣接リスト
  - 2つのノードの隣接を判断するには、隣接リスト内のノードを全てチェックする必要がある
  - あるノードに隣接するノードを探索するには、隣接リスト内のノードのみをチェックすればよい
  - 疎なグラフ（ノードに対してリンクは少ない）を効率的に表現できる

グラフの実装例

- 先ほどの隣接リストの考え方を発展させて、値をもつノードからなる有向グラフを、隣接リストで表現することを考えます
  - ※隣接リストではリンクリストを用いるケースもよくありますが、ここではPythonのリストを用いています
- 各ノードに隣接するノードへのリンクのリストを持たせます
  - リンクリストの場合や木構造の場合と似た考え方です

```
class Node:
    def __init__(self, value):
        self.value = value
        self.links = []

    def __str__(self):
        return str(self.value)
```



グラフの実装例

- 使いやすいように、いくつかのメソッドを書き足します
  - また探索のための属性も定義しておきます

```
class Node:
    def __init__(self, value):
        self.value = value
        self.traversed = False
        self.links = []

    def __str__(self):
        return str(self.value)

    def add_link(self, node):
        self.links.append(node)

    def adjacent(self, node):
        return node in self.links
```

value : ノードの値
traversed : 訪問の有無(後で使います)
links : 隣接するノードのリスト

このノードに隣接するノードを追加する

他のノードとの隣接を判定する

グラフの実装例

- 以下のコードで、どのようなグラフになるか分かりますか？

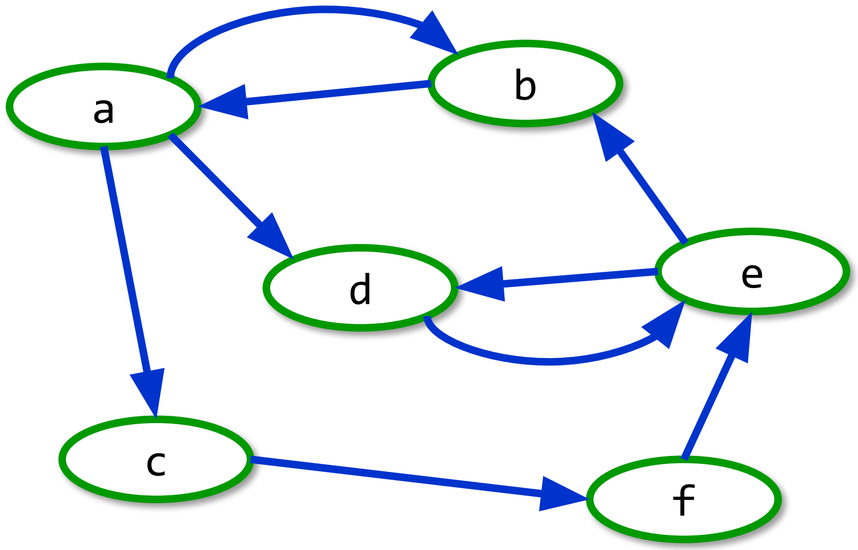
```
a = Node('a')
b = Node('b')
c = Node('c')
d = Node('d')
e = Node('e')
f = Node('f')

a.add_link(b)
a.add_link(c)
a.add_link(d)
b.add_link(a)
c.add_link(f)
d.add_link(e)
e.add_link(b)
e.add_link(d)
f.add_link(e)

graph = [a, b, c, d, e, f]
```

ノードを生成する

リンクを追加する




```
a.adjacent(b)
```

```
True
```