Utiliser Python pour tracer des graphiques depuis un code C ou C++

Hélène Toussaint, octobre 2020

But de ce tutoriel : ce document s'adresse aux développeurs C/C++ qui souhaitent générer de manière automatique des dessins (notamment des représentations graphiques de données) à l'aide de Python à partir de leur code C/C++. Les exemples fournis sont extraits de problèmes de recherche opérationnelle et permettent de représenter les données d'entrée et / ou solutions de divers problèmes d'optimisation.

A qui s'adresse ce tutoriel ? : développeurs C/C++

Prérequis : Connaissances en C ou C++, connaissances élémentaires de Python (installation de Python et des modules Python, syntaxe de base : boucles, listes, matrices)

Table des matières

1 Appeler Python depuis un code C/C++ 2
1.1 Généralités 2
1.2 Méthodologie 2
2 Ecriture du script Python 3
2.1 Exemple complet 3
2.2 ETAPE 1 : récupérer le nom du fichier de données passé en argument 4
2.3 ETAPE 2 : lire le fichier 5
2.4 ETAPE 3 : représentation graphique 6
3 Quelques exemples de représentations graphiques 6
3.1 Représenter des points du plan 2D et des tournées 7
3.2 Représenter des variations en fonction du temps 8
3.3 Représenter des placements 2D (*bin packing*) 9
3.4 Tracer des graphes avec pydotplus / Graphviz 10
4 Références 12

1 Appeler Python depuis un code C/C++

1.1 Généralités

Il existe deux manières d'appeler Python depuis un code C/C++ :

- avec un appel à la fonction `int system(const char* command) [1]` : l'échange de données entre le C/C++ et Python s'effectue alors par l'intermédiaire d'un fichier texte qui est écrit depuis C/C++ et lu depuis Python. La fonction **system** se charge de lancer la commande Python, avec en paramètre, le script de son choix ;

- en utilisant l'API Python/C ([2]). L'échange de données s'effectue directement au niveau du code et Python est appelé à l'aide des fonctions définies dans l'API.

Dans ce document on utilise la première méthode qui est, à mon avis, beaucoup plus simple que la seconde car :

1. elle ne nécessite pas d'apprendre à utiliser l'API Python/C,
2. le code C/C++ est compilé de manière habituelle (pas besoin de *linker* avec l'API Python/C),
3. le code pour générer les graphiques est écrit directement en Python et il existe une multitude d'exemples sur internet pour créer des dessins en tout genre avec Python (mais très peu d'exemples qui utilisent l'API Python/C).

L'inconvénient de cette méthode est l'écriture d'un fichier texte auxiliaire pour échanger les données entre le C/C++ et Python. L'écriture et la lecture de ce fichier nécessite un accès au disque ce qui peut dégrader les temps de calcul de notre programme C/C++ si on le fait de nombreuses fois. Cependant, le but ici étant simplement de représenter graphiquement des données, il est fort probable qu'on ait à écrire et lire ce fichier auxiliaire seulement une fois, ce qui aura un impact négligeable sur les temps de calcul.

1.2 Méthodologie

Prérequis : Python doit être installé sur votre PC et le chemin vers Python doit être ajouté à la variable d'environnement *PATH.* Normalement la variable d'environnement *PATH* est mise à jour automatiquement lors de l'installation de Python (sous Windows veillez à bien cocher « Add Python to PATH » lors de l'installation).

L'exemple de code suivant montre comment exécuter un script Python (`dessine.py`), qui accepte en entrée un fichier texte (`data.txt`), depuis un code C ou C++ en utilisant la fonction **system**().

```
int main()
{
	//lance le script python dessine.py qui accepte un fichier texte en entrée
	system("python dessine.py data.txt");

	return 0;
}
```

Il reste maintenant à voir comment écrire le script `dessine.py` de sorte qu'il lise le fichier texte en entrée et dessine ce que l'on souhaite. C'est l'objet de la section suivante.

2 Ecriture du script Python

On va voir dans cette section comment écrire un script Python qui permette de tracer des graphiques en utilisant des données contenues dans un fichier texte passé en argument de la ligne de commande. Afin de donner au lecteur une idée générale de la forme d'un tel script on commencera par présenter un exemple complet (mais minimal) dans une première sous-section, les sous-sections suivantes donneront des explications et des détails sur chaque partie du script.

2.1 Exemple complet

Le script doit comprendre 3 étapes principales :

1. récupérer le nom du fichier de données passé en argument de la ligne de commande,
2. extraire les données du fichier,
3. représenter graphiquement les données.

Le script Python suivant (nommé **`dessine.py`**) lit un fichier qui contient une liste de points identifiés par leurs coordonnées (x, y) et les trace dans le plan 2D.

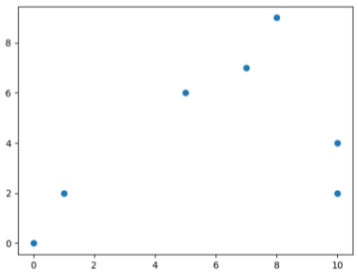
```
import sys
import os
import numpy as np
import matplotlib.pyplot as plt

#-----------------------------------------------------------------------------------------
# ETAPE 1 : RECUPERER LE NOM DU FICHIER DE DONNEES PASSE EN ARGUMENT DE LA LIGNE DE COMMANDE

# 1.1. on verifie qu'on a le bon nombre d'arguments
if len(sys.argv) != 2:
        print("il faut un parametre en ligne de commande : le nom de l'instance")
        sys.exit()

#1.2. on recupère le nom du fichier de données
nomFic = sys.argv[1]

#-------------------------------------------
# ETAPE 2 : EXTRAIRE LES DONNEES DU FICHIER

#2.1. on vérifie que le chemin vers le fichier est correct
if not os.path.exists(nomFic):
        print("ERREUR Ouverture fichier " + nomFic)
        sys.exit()

#2.2. on stocke le contenu du fichier dans une matrice
M = np.loadtxt("data.txt")

#-------------------------------------------
# ETAPE 3 : DESSINER ET SAUVEGARDER LE DESSIN

#3.1. on dessine le nuage de points
plt.scatter(M[:,0], M[:,1])
```

```
#3.2. on sauvegarde le dessin en png
plt.savefig("graph.png")

#3.3. facultatif : affiche le dessin à l'écran (ne pas mettre avant savefig)
plt.show()
```

Contenu du fichier **dessine.py**

```
0 0
1 2
10 4
5 6
8 9
10 2
7 7
```

Contenu du fichier de données **data.txt**



Résultat du script : **graph.png**

Remarque : Ce script nécessite 4 modules. Tous les modules Python ne sont pas installés en même temps que Python (il y en a beaucoup trop). Il faut donc installer ceux que l'on souhaite utiliser à la main. Cela se fait très facilement à l'aide de l'outil **pip** (qui s'installe automatiquement en même temps que Python).

L'utilisation classique de cet outil en ligne de commande est la suivante (pour plus de détails voir [4]) :

```
python -m pip install nomDuModule
```

Exemple :

```
python -m pip install numpy
```

2.2 ETAPE 1 : récupérer le nom du fichier de données passé en argument

Pour pouvoir lire des arguments passés en ligne de commande à un script Python il est nécessaire d'utiliser le module **sys** [3]. Pour cela on écrit en tête de notre script la directive **import sys** : la directive **import** est plus ou moins l'équivalent du **#include** en C (une grosse différence néanmoins c'est que toutes les fonctions du module **sys** utilisées dans notre script devront être préfixées par **sys.** qui agit comme un espace de noms).

Les différents arguments sont contenus dans la variable **sys.argv** qui est une liste de chaînes de caractères (exactement comme en C). La première chaîne de caractères (**sys.argv[0]**) est le nom du script (dans notre exemple précédent ça serait donc **dessine.py**). La variable **sys.argv[1]** contient le premier argument de la ligne de commande (dans notre exemple c'est le nom du fichier de données).

On récupère donc le nom du fichier de données par l'instruction

```
nomFic = sys.argv[1]
```

Une bonne pratique consiste à vérifier, avant d'accéder au contenu de `sys.argv[1]`, qu'on a le nombre d'arguments attendus passés en ligne de commande (dans notre exemple : 2) :

```
if len(sys.argv) != 2:
        print("il faut un parametre en ligne de commande : le nom de l'instance")
        sys.exit()
```

L'instruction `sys.exit()` permet de mettre fin au script de manière prématurée.

2.3 ETAPE 2 : lire le fichier

Cas 1 : Le fichier contient uniquement une matrice

Si le fichier contient uniquement une matrice, le module **numpy** [5] offre une fonction qui permet de charger directement le contenu du fichier dans une matrice :

```
M = np.loadtxt("data.txt")
```

Remarquons l'utilisation de **np.** au lieu de **numpy.**. Cela est possible car on a chargé le module **numpy** en lui donnant l'alias **np** (**import numpy as np**). C'est une pratique très courante, surtout pour les modules qui ont un nom assez long.

Avant de lire le fichier, une bonne pratique consiste à vérifier que le fichier existe bien. Pour cela on utilise le module **os.path** [6] :

```
if not os.path.exists(nomFic):
       print("ERREUR Ouverture fichier " + nomFic)
       sys.exit()
```

Cas 2 : Le fichier contient des lignes de natures différentes

Si le fichier contient des lignes de natures différentes (i.e. différents types, longueurs différentes, …) alors il n'est plus possible d'utiliser **np.loadtxt().** Dans ce cas une technique possible consiste à ouvrir le fichier, lire son contenu ligne par ligne et placer le résultat dans une liste de chaîne de caractères où chaque chaîne de caractères représente une ligne du fichier. Naturellement on pensera bien à fermer le fichier après la lecture (comme en C !) :

```
fp = open(nomFic)              # ouverture du fichier
lines = fp.read().split("\n") # lecture ligne par ligne
fp.close()                     # fermeture du fichier
```

Chaque ligne est ainsi stockée sous forme de chaine de caractères. Si une ligne représente un tableau de valeurs il faut alors transformer la chaine correspondante en tableau de valeurs. Imaginons que ce soit des entiers, une possibilité pour transformer une chaine de caractères en tableau d'entiers est la suivante :

```
tab = [int(x) for x in chaine.split(' ')]
```

La ligne de code précédente découpe la chaine de caractères **chaine** en utilisant l'espace (**' '**) comme délimiteur et transforme chaque élément en entier. Le résultat est stocké dans le tableau **tab**.

Exemple

Dans l'exemple suivant un fichier de données contient un tableau de 6 flottants sur une ligne et un tableau de 9 entiers sur une seconde ligne.

```
fp = open('data.txt')
lines = fp.read().split("\n")
fp.close()

tab1 = [float(x) for x in lines[0].split(' ')]
tab2 = [int(x) for x in lines[1].split(' ')]
```

Code pour lire le fichier `data.txt` et stocker chaque ligne comme un tableau de valeurs

```
1.5 6.8 10.2 6 0.3 8
1 2 3 9 8 7 10 12 16
```

Contenu du fichier data.txt

2.4 ETAPE 3 : représentation graphique

Python offre une très grande variété de représentations graphiques notamment à l'aide du module **matplotlib** [7]. La manière la plus simple d'utiliser ce module est via les fonctions définies dans **matplotlib.pyplot**, c'est pourquoi on charge en général directement **matplotlib.pyplot** à l'aide la directive **import matplotlib.pyplot as plt**.

Chaque fonction de **matplotlib** accepte de nombreux paramètres facultatifs qui permettent de personnaliser le rendu du dessin (couleurs, formes, taille, …). Le site python-simple [8] offre une très bonne synthèse des différentes possibilités.

Quel que soit la représentation graphique choisie, deux fonctions de **matplotlib** sont souvent utiles : **savefig("nomDuFichier.png")** qui permet de sauvegarder sur disque le dessin dans un fichier **png** et **show()** qui permet d'afficher la figure à l'écran.

On trouve sur internet une multitude d'exemples et de documentations concernant l'utilisation de **matplotlib**. On présentera donc dans la section suivante seulement quelques exemples qui peuvent être utiles pour représenter des entrées / sorties de différents types de problème d'optimisation mais sans entrer dans les détails des fonctions utilisées (pour plus de détails le lecteur est invité à consulter les sites [7] et [8]).

3 Quelques exemples de représentations graphiques

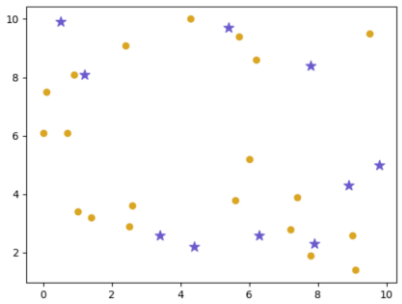
Dans cette section on donne quelques exemples de représentations graphiques qui peuvent être utiles dans le cadre de programmes de recherche opérationnelle (tournée de véhicules, ordonnancement, *packing*, …). Les codes présentés ici correspondent à l'étape 3 (section 2.4), on ne rappelle pas les étapes 1 et 2 qui sont toujours les mêmes.

Dans un souci de lisibilité des exemples et afin de se concentrer uniquement sur la partie graphique, les données sont créées « à la main » (sans lire de fichier).

3.1 Représenter des points du plan 2D et des tournées

Représentation de 2 ensembles de points

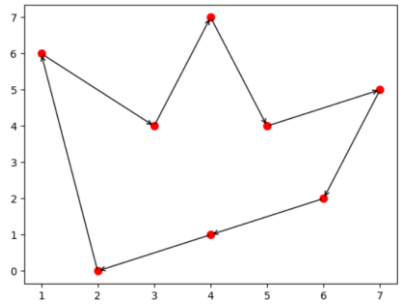
```
import numpy as np
import random as rd
import matplotlib.pyplot as plt

X1 = [rd.randint(0,100)/10 for i in range(20)]
Y1 = [rd.randint(0,100)/10 for i in range(20)]

X2 = [rd.randint(0,100)/10 for i in range(10)]
Y2 = [rd.randint(0,100)/10 for i in range(10)]

plt.scatter(X1, Y1, color = "goldenrod")
plt.scatter(X2, Y2, color = "slateblue", marker = '*', s=100)

plt.show()
```



Représentation d'une tournée : ajout de flèches

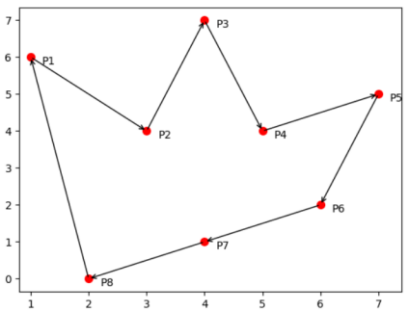
On détourne l'utilisation de la fonction **`plt.annotate`** [9] qui sert normalement à annoter le schéma.

```
import matplotlib.pyplot as plt

X = [1,3,4,5,7,6,4,2]
Y = [6,4,7,4,5,2,1,0]

plt.scatter(X, Y, color = "red", s = 50)

i = 0
N = len(X)
while i < N:
    suiv = (i+1)%N
    plt.annotate(text='', xy=(X[i],Y[i]), xytext =
          (X[suiv],Y[suiv]), arrowprops=dict(arrowstyle='<-'))
    i=i+1
plt.show()
```



Représentation d'une tournée : ajout de labels sur les points

```
import matplotlib.pyplot as plt

X = [1,3,4,5,7,6,4,2]
Y = [6,4,7,4,5,2,1,0]
label = ['P1', 'P2', 'P3', 'P4', 'P5', 'P6', 'P7', 'P8', 'P9']
eps = 0.2 #decalage du label par rapport au point

plt.scatter(X, Y, color = "red", s = 50)

i = 0
N = len(X)
while i < N:
    suiv = (i+1)%N
    plt.annotate(text='', xy=(X[i],Y[i]), xytext =
         (X[suiv],Y[suiv]), arrowprops=dict(arrowstyle='<-'))
    i=i+1

for i in range(len(X)):
    plt.text(X[i]+eps, Y[i]-eps, label[i])

#on agrandit un peu les limites sur X pour que les
# labels ne dépassent pas le cadre
plt.xlim(min(X)-eps, max(X)+2*eps)

plt.show()
```



3.2 Représenter des variations en fonction du temps

Courbe

```
import matplotlib.pyplot as plt

Y = [10, 15, 2, 6, 16, 7, 19, 21, 4]
X = range(len(Y))

plt.plot(X, Y, marker = 'o')

plt.xlabel("unité de temps")
plt.ylabel("quantité de produit")
plt.title("quantité de produit par unité de temps")

plt.show()
```

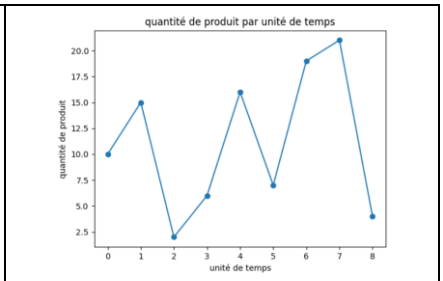


Diagramme en bâtons

```
import matplotlib.pyplot as plt

Y = [10, 15, 2, 6, 16, 7, 19, 21, 4]

# dans un diagramme en bar l'axe X doit être une liste
# de chaines de caractères
i = 0
X = []
while i < len(Y):
    X.append(str(i))
    i=i+1

plt.bar(X, Y)
plt.xlabel("période")
plt.ylabel("quantité de produit")
plt.title("quantité de produit disponible par période")

plt.show()
```

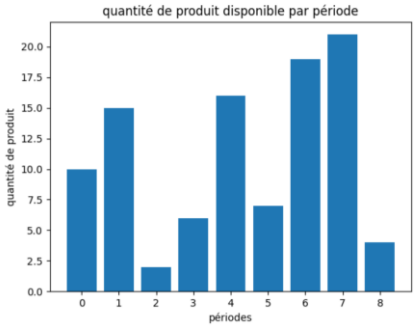
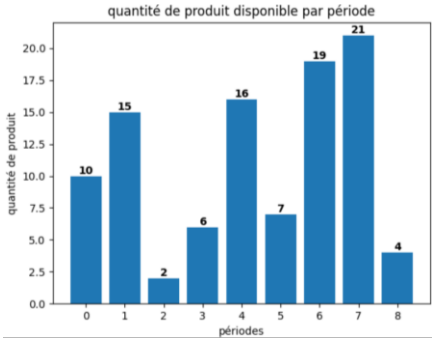


Diagramme en bâtons: ajout des valeurs sur le graphique

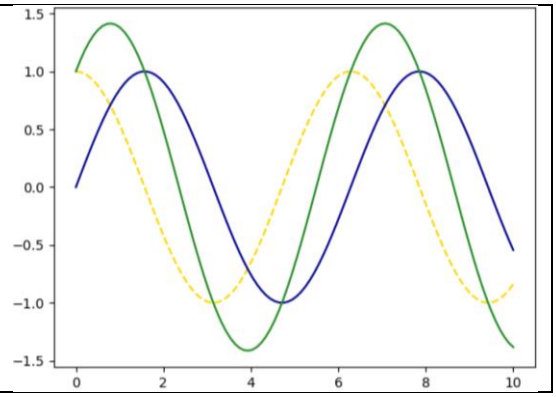
```
import matplotlib.pyplot as plt

Y = [10, 15, 2, 6, 16, 7, 19, 21, 4]

# dans un diagramme en batons l'axe X doit être une liste
# de chaines de caractères
i = 0
X = []
while i < len(Y):
    X.append(str(i))
    i=i+1

bar_plot = plt.bar(X, Y)
plt.xlabel("périodes")
plt.ylabel("quantité de produit")
plt.title("quantité de produit disponible par période")

#ajout de texte au dessus des batons
for i,rect in enumerate(bar_plot):
    h = rect.get_height()
    plt.text(rect.get_x() + rect.get_width()/2., h,
         str(Y[i]), ha='center', va='bottom', weight="bold")

plt.show()
```



Tracer plusieurs graphiques sur une même figure

On peut soit tracer les graphiques dans un même repère…

```
import matplotlib.pyplot as plt
import numpy as np


#genere 20 points dans l'intervalle [0,10]
X = np.linspace(0,10,100)
Y1 = np.cos(X)
Y2 = np.sin(X)
Y3 = Y1+Y2

plt.plot(X, Y1, '--', color = "gold")
plt.plot(X, Y2, color = "darkblue")
plt.plot(X, Y3, color = "forestgreen")

plt.show()
```

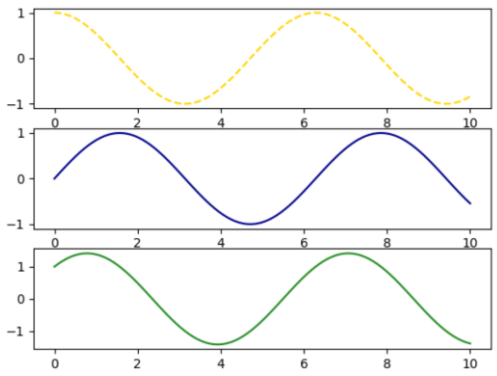


… soit utiliser des sous figures à l'aide de la fonction **`subplot(nlig, ncol, nfig)`**, où :

- **`nlig`** est le nombre de sous-figures placées verticalement (les unes en dessous des autres)
- **`ncol`** est le nombre de sous-figures placées horizontalement
- **`nfig`** est le numéro de la sous-figure (la numéro 1 sera placée le plus en haut à gauche)

Toutes les instructions écrites après l'appel à **`subplot`** concernent la sous-figure correspondante.

```
import matplotlib.pyplot as plt
import numpy as np


#genere 20 points dans l'intervalle [0,10]
X = np.linspace(0,10,100)
Y1 = np.cos(X)
Y2 = np.sin(X)
Y3 = Y1+Y2

plt.subplot(3,1,1)
plt.plot(X, Y1, '--', color = "gold")

plt.subplot(3,1,2)
plt.plot(X, Y2, color = "darkblue")

plt.subplot(3,1,3)
plt.plot(X, Y3, color = "forestgreen")

plt.show()
```

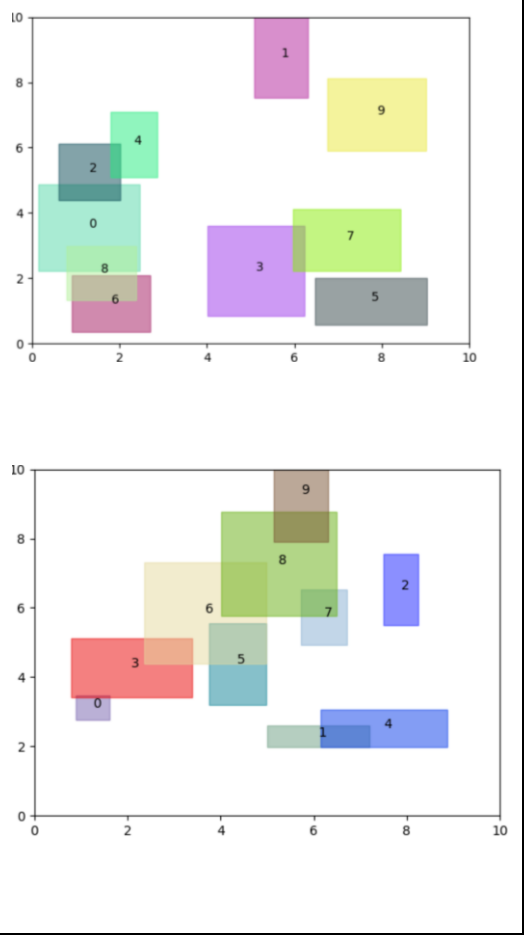


3.3 Représenter des placements 2D (*bin packing*)

On souhaite dessiner un ensemble de rectangles dans le plan 2D. On utilise pour cela la classe **`Patch`** de **`matplotlib`** (**`matplotlib.patches.Patch`**), qui sert à dessiner des objets géométriques (lignes, rectangles, ellipses, …).

L'exemple suivant définit une classe pour stocker les informations sur les rectangles (position du coin inférieur gauche et dimensions). On pourrait bien sûr se passer de la classe et stocker les différentes informations dans de simples listes. Les positions, dimensions et couleurs des rectangles sont générées aléatoirement.

```
import matplotlib.pyplot as plt
import matplotlib
import random as rd

#on crée une classe pour stocker les attributs d'un
rectangle
class MyRectangle:
    x = 0
    y = 0
    largeur = 1
    hauteur = 1
    couleur = (0,0,0)


#on crée une liste de 10 rectangles
l_rect = []
for i in range(10):
    rect = MyRectangle()
    rect.x = rd.uniform(0,8)
    rect.y = rd.uniform(0,8)
    rect.largeur = rd.uniform(0.5,3)
    rect.hauteur = rd.uniform(0.5,3)
    rect.couleur = (rd.random(), rd.random(),
                    rd.random())
    l_rect.append(rect)

#affiche les rectangles
axes = plt.gca()
for i in range(10):
    rect = l_rect[i]
    axes.add_artist(matplotlib.patches.Rectangle(
        (rect.x, rect.y), rect.largeur, rect.hauteur,
         color = rect.couleur, alpha = 0.5))
    plt.text(rect.x + rect.largeur / 2.,
         rect.y + rect.hauteur / 2, str(i))

#fixe les limites du cadre (par défaut [0,1]*[0,1])
plt.xlim(0,10)
plt.ylim(0,10)

plt.show()
```



3.4 Tracer des graphes avec pydotplus / Graphviz

Un outil très intéressant pour représenter des graphes est **Graphviz** [10]. Une manière très simple de l'utiliser depuis un code C/C++ c'est de passer par l'écriture d'un fichier texte qui décrit le graphe, puis d'appeler **Graphviz** en ligne de commande depuis le code C/C++ à l'aide de la fonction **`system()`** du C (comme on l'a fait pour Python dans la section 1.2).

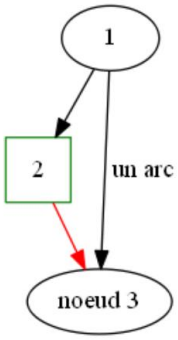
L'utilisation de **Graphviz** en ligne de commande est très bien expliquée dans la documentation de **Graphviz** (voir https://graphviz.org/pdf/dotguide.pdf).

On s'intéresse ici au module **`pydotplus`** [11] qui offre une interface Python à **Graphviz** (plus précisément à l'outil **dot** de **Graphviz**). L'utilisation de ce module est tout aussi simple que l'écriture

du fichier texte. Remarquons qu'il n'y pas vraiment d'intérêt à utiliser **pydotplus** au lieu d'utiliser directement **Graphviz** depuis un code C/C++ sauf si on souhaite inclure le graphe dans une figure Python (avec d'autres éléments graphiques).

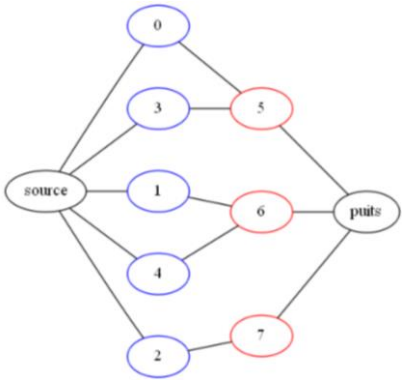
L'exemple suivant montre la syntaxe minimale pour créer et dessiner un graphe.

```
import pydotplus as pydot

#1. on crée le graphe : digraph -> graphe orienté
graph = pydot.Dot(graph_type="digraph")

#2. on ajoute des sommets au graphe
node1 = pydot.Node('1')
node2 = pydot.Node('2', shape = "square", color = "forestgreen")
node3 = pydot.Node('3', label="noeud 3")

graph.add_node(node1)
graph.add_node(node2)
graph.add_node(node3)

# 3 on cree des arcs
graph.add_edge(pydot.Edge(node1, node2))
graph.add_edge(pydot.Edge(node1, node3, label = " un arc"))
graph.add_edge(pydot.Edge(node2, node3, color="red"))

#sauvegarde du graphe dans un fichier png
graph.write_png("monGraphe.png")
```



Il n'y a actuellement, à ma connaissance, pas de documentation très détaillée de **pydotplus**. Cependant l'utilisation est assez intuitive si on connait déjà un peu l'outil **dot** de Graphviz. Chaque élément du graphe est personnalisable à l'aide de paramètres. Les noms des paramètres et la valeur qu'ils peuvent prendre sont exactement ceux des attributs décrits dans la documentation de Graphviz (voir [12] pour la liste détaillée des attributs).

```
#graphe non orienté et dessiné horizontalement
graph = pydot.Dot(graph_type="graph", rankdir="LR")

source = pydot.Node('s', label="source")
puits = pydot.Node('p', label="puits")
graph.add_node(source)
graph.add_node(puits)

listNode = []
for i in range(5):
        node = pydot.Node(str(i), color="blue")
        graph.add_node(node)
        listNode.append(node)
        graph.add_edge(pydot.Edge(source, node))

listNode2 = []
for i in range(3):
        node = pydot.Node(str(i+5), color="red")
        graph.add_node(node)
        listNode2.append(node)
        graph.add_edge(pydot.Edge(node, puits))

for i in range(5):
        graph.add_edge(pydot.Edge(listNode[i], listNode2[i%3]))

graph.write_png("monGraphe.png")
```



4 Références

[1] http://www.cplusplus.com/reference/cstdlib/system/
[2] https://docs.python.org/fr/3.8/c-api/index.html#c-api-index
[3] https://docs.python.org/fr/3/library/sys.html
[4] https://docs.python.org/fr/3.8/installing/index.html
[5] https://numpy.org/doc/stable/contents.html#numpy-docs-mainpage
[6] https://docs.python.org/3/library/os.path.html
[7] https://matplotlib.org/
[8] https://python-simple.com/python-matplotlib/matplotlib-intro.php
[9] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.annotate.html
[10] https://graphviz.org/
[11] https://pydotplus.readthedocs.io/reference.html#module-pydotplus.graphviz
[12] https://graphviz.org/doc/info/attrs.html