Programmation orientee objet et interfaces graphiques

April 2, 2021

1 Notion d'objet en Python

Pour illustrer la notion d'objet en python, nous allons analyser ce que fait une courte suite d'instruction.

[ ]:
```
a = [0]
b = [0]
a[0] = 1
b[0] = a[0]
print(b)
a[0] = 2
print(b)
```

On obtient bien le résultat attendu en raisonnant de la manière suivante :

[ ]:
```
a = [0] # On initialise a comme [0]
b = [0] # On initialise b comme [0]
a[0] = 1 # On modifie a comme [1]
b[0] = a[0] # On modifie b comme a soit [1]
print(b) # On affiche la valeur courante de b soit [1]
a[0] = 2 # On modifie a comme [2]
print(b) # Mais b n'a pas changé donc on affiche toujours [1]
```

Appliquons ce même raisonnement aux lignes de commandes suivantes

[ ]:
```
a = [1] # On initialise directement a comme [1]
b = a # On initialise directement b comme a donc [1]
print(b) # On affiche b donc [1]
a[0] = 2 # On modifie a comme [2]
print(b) # Mais b n'a pas changé donc on affiche toujours [1]
```

On ne trouve pas le résultat attendu cette fois. On a l'impression que la modification de a a entrainé automatiquement la modification de b. C'est parce qu'il faut raisonner comme dans la vie de tout les jours en terme d'objet.

[ ]:
```
a = [1] # a est l'objet [1]
b = a # b est le même objet que a
print(b) # On affiche b donc on affiche a soit [1]
a[0] = 2 # On modifie a le premier élément de a soit de l'objet [1] qui devient␣
↪[2]
```

```
print(b) # Mais b n'a pas changé, b est toujours le même objet que a, soit␣
 ↪maintenant [2]
```

Si on initialise b comme un autre objet indépendament de a, on retrouvera alors qu'une modification de a, ne change pas l'objet b. Pour cela il faut écrire :

[ ]:
```
a = [1] # a est un objet [1]
b = [1] # b est un autre objet qui vaut aussi [1]
print(b) # On affiche b soit [1]
a[0] = 2 # On modifie l'objet a pour [2]
print(b) # b est toujours le même objet, [1].
```

Lorsqu'on définit a et b comme les mêmes objets on a observé que toutes modifications de a entraîne une modification de b. Il en est de même aussi pour toute modification de b entraîne une modification de a.

[ ]:
```
a = [1]
b = a
print(b)
b[0] = 2
print(a)
```

Revenons sur la première séries d'instruction et interprétons-la en terme d'objet.

[ ]:
```
a = [0] # a est un objet [0]
b = [0] # b est un autre objet [0]
a[0] = 1 # on modifie le premier élément de l'objet a pour [1]
b[0] = a[0] # on modifie le premier élément de l'objet b par le premier élément␣
 ↪de l'objet a soit [1],
            # mais ces deux objets restent indépendants.
print(b) # on affiche l'objet b soit [1]
a[0] = 2 # on modifie le premier élément de l'objet a pour [2]
print(b) # b est toujours le même objet [1]
```

L'interprétation en terme d'objet est donc plus proche d'une interprétation d'un objet "réel" que l'interprétation habituelle qu'on a faite des variables en informatique.

Jusqu'à présent on considère les variables (ici a et b) comme des espaces mémoires séparés et l'opération a = qqch comme l'affectation de qqch dans l'espace mémoire de a. En programmation orienté objet les variables sont des noms que l'on donne à un objet. Lorsque l'on écrit a = un élément comme par exemple a = [1], on définit un objet [1] et on le nomme par la variable a. Quand on écrit l'égalité entre deux noms d'objet par exemple a = b, on définit que ces deux noms désignent le même objet, donc a et b pointent vers le même espace mémoire. Alors si on modifie l'espace mémoire en utilisant le nom a ou b le résultat est le même.

1.1 exercices

Déterminer le résultat des cellules suivantes.

[ ]:
```
a = 3
b = a + 4
c = a
a = b
```

```
c = 3.14

print(a,b,c)
```

[ ]:
```
a = [2,3,7]
b = a
print(b[1])
a[1] = 4
print(b[1])
a = [5,6,7,8]
print(b[1])
```

[ ]:
```
def exemple(arg):
    print(arg[1])
    arg[1] = 4
    print(arg[1])
    arg = [5,6,7,8]

a = [1,2,3,4]
exemple(a)
print(a[1])
```

[ ]:
```
a = [1,2,34,45]
b = a
c = a[1]
a[2] = 1
a[1] = 5
print(b[2]+c)
```

2 Classes, attributs, et méthodes

Pour rendre les objets plus facile d'utilisation, par l'utilisateur d'un programme informatique on définit trois notions propres aux objets (classe, attributs, et méthodes), qui les raprochent d'objet réels.

Une classe est un groupe d'objet qui vont partager des points communs. C'est donc un ensemble dans lequel on va classer des objets. On peut chercher par exemple à remplir son annuaire. On va donc définir une classe d'objet que sont les personnes avec :

[ ]:
```
class Personne():
    """Classe des personnes"""
    nom = ""
```

On peut alors définir un nouvel objet de cette classe avec la commande : nom_de_variable = nom_de_classe()

[ ]:
```
le_prof = Personne()
```

On vient alors de définir que le_prof est une personne ou plutôt le_prof est un objet appartenant à la classe d'objet Personne. Testons quel est le type de l'objet le_prof

[ ]:
```
type(le_prof)
```

Python nous répond bien qu'il s'agit d'un objet de la classe Personne.

On peut vouloir maintenant donner des informations sur un objet de la classe Personne, par exemple son nom et son prénom. Pour cela on utilise des attributs, on remarque que l'attribut nom est déjà définit à l'initialisation d'un objet de la classe Personne comme une chaîne de caractère vide.

On peut appeler l'attribut nom d'un objet grace à la commande objet.attribut, par exemple l'attribut nom de le_prof est une chaine de caractère vide.

[ ]:
```
print(le_prof.nom)
```

On peut ensuite modifier cet attribut de l'objet en lui assignant une autre valeur.

[ ]:
```
le_prof.nom = "Bisognin"

print(le_prof.nom)
```

On peut ensuite rajouter d'autres attributs, soit à la définition de la classe, comme:

[ ]:
```
class Personne():
    """Classe des personnes"""
    nom = ""
    prenom = ""
```

[ ]:
```
le_prof.prenom = "Rémi"

print(le_prof.prenom, le_prof.nom)
```

soit directement en affectant une valeur à cet attribut, comme :

[ ]:
```
le_prof.email = "remi.bisognin@ac-versailles.fr"

print(le_prof.prenom, le_prof.nom, le_prof.email)
```

on remarque que c'est une façon très explicite de stocker les informations comparée à une structure de données comme une liste où il faut se souvenir que l'indice 0 correspond au nom, l'indice 1 au prénom, ...

On peut enfin définir des méthodes pour chaque classe d'objet. Une méthode est une fonction définie pour une classe qui prend en argument un objet avec ses attributs et peut aussi prendre d'autres arguments. L'utilisateur n'a alors plus à se poser la question de quelle fonction écrire, c'est lors de la définition de la classe que la fonction adaptée à l'objet a été définie.

Lorsqu'une fonction est définie dans une classe, on parle alors de méthode. Cette méthode est une fonction qui prend en argument l'objet lui même qui est noté par convention self et peut prendre d'autre argument si besoin.

Par exemple on peut définir la méthode qui dit bonjour à une personne avec :

[ ]:
```
class Personne():
    """Classe des personnes"""
    nom = ""
    prenom = ""
    le_prof.email = "remi.bisognin@ac-versailles.fr"

    def bonjour(self):
        print("Bonjour", self.prenom, self.nom, '!')
```

Pour appeler une méthode sur un objet self on utilise alors la commande self.methode()

[ ]:
```
le_prof = Personne()
le_prof.nom = "Bisognin"
le_prof.prenom = "Rémi"
le_prof.bonjour()
```

Lorsque les méthodes sont correctement définies dans les classes, elle sont ensuite plus facile d'utilisation car lorsqu'on veut faire une action avec un objet, on possède une liste de méthode qui réalise l'action appropriée pour chaque type d'objet.

2.1 exercice

On souhaite manipuler la liste des amis d'une personne dans la classe Personne. Créer les attributs correspondant, puis les méthodes pour ajouter un ami et afficher un ami.

Tester votre classe en créant une suite de Personne avec des liens d'amitiés.

Ecrire une méthode qui permet de mettre à jour automatiquement les attributs d'une personne de manière à ce que : "les amis de mes amis soient mes amis."

[ ]:
```

```

Certaines méthodes spéciales existent avec des rôles prédéfinis.

La méthode __init__ sera appelée automatiquement lors de la création de l'objet. On peut l'utiliser pour initialiser des attributs.

[ ]:
```
class Personne():
    def __init__(self, nom="", prenom=""): #Initialisation avec possiblité de␣
↪passer des attributs en argument.
                                    #En l'absence d'argument les valeurs par␣
↪défaut sont ici "" et ""
        self.nom = nom
        self.prenom = prenom
    def bonjour(self):
        print("Bonjour", self.prenom, self.nom, '!')
```

[ ]:
```
le_prof = Personne("Bisognin", "Rémi")
le_prof.bonjour()
```

La méthode __repr__ sert a représenter l'objet sous forme d'une chaîne de caractère par la fonction print.

En effet si on teste la fonction print sur l'objet ou l'objet directement on obtient un résultat peu lisible :

[ ]:
```
print(le_prof)
```

[ ]:
```
le_prof
```

On peut vouloir modifier cette fonction pour afficher le nom complet

[ ]:
```
class Personne():
    def __init__(self, nom="", prenom=""):
        self.nom = nom
        self.prenom = prenom
    def bonjour(self):
```

```
        print("Bonjour", self.prenom, self.nom, '!')
    def __repr__(self):
        return self.prenom + " " + self.nom
```

[ ]:
```
le_prof = Personne("Bisognin", "Rémi")
print(le_prof)
```

[ ]:
```
le_prof
```

Il existe aussi la possibilité de redéfinir des méthodes existantes, par exemple la méthode def __add__(self,other): sera appelée par la commande self+other. La méthode def __or__(self,other): sera appelée par la commande self | other.

2.2 exercice

Ré-écrire la méthode d'ajout des amis tel que l'opération + ajoute les amis de deux personnes. Soit personne3 = personne2 + personne1, donne la personne3 est la personne2 avec les amis de la personne1 en plus.

[ ]:
```
```

Enfin nous avons vu comment définir une classe d'objet. Dans cette classe, nous avons défini l'ensemble des attributs et des méthodes. Il est cependant possible d'hériter des méthodes d'une autre classe - ce qui permet d'avoir plusieurs classes possédant des méthodes identiques.

Continuons notre exemple de personne: suivant la nationalité, je souhaite pouvoir écrire un message de bonjour différent. Pour cela, je vais créer une classe PersonneFrancaise et PersonneAnglaise qui hérite des attributs et méthodes de la classe Personne, mais qui ont chacune une méthode bonjour() adaptée à la nationnalité de la personne.

[ ]:
```
class Personne():
    def __init__(self, nom="", prenom=""):
        self.nom = nom
        self.prenom = prenom
        self.liste_ami = set()
    def __repr__(self):
        return self.prenom + " " + self.nom

class PersonneAnglaise(Personne):
    def bonjour(self):
        print("Hello", self.prenom, self.nom, '!')

class PersonneFrancaise(Personne):
    def bonjour(self):
        print("Bonjour", self.prenom, self.nom, '!')
```

[ ]:
```
la_prof = PersonneAnglaise(prenom="Mme", nom="Guilheneuf")
la_prof.bonjour()
```

[ ]:
```
le_prof = PersonneFrancaise(prenom="Rémi", nom="Bisognin")
le_prof.bonjour()
```

Cette propriétée d'hériatage permet d'éviter à l'auteur du programme d'avoir à recopier l'ensemble des attributs et méthodes pré-existante.

Et il permet à l'utilisateur de ne pas avoir besoin de savoir si le_prof ou la_prof est anglais ou français, .bonjour() sera toujours la méthode adaptée à l'objet.

3 Exercice : une classe de vecteur

On souhaite effectuer des calculs vectoriel en 3D en python. Définir une classe d'objet vecteur qui représente un vecteur par une liste de ses trois composantes. On pourra hériter des propriétés des objets de la classe list et définir des méthodes de calcul vectoriel comme le produit scalaire, la norme, les produits scalaires et vectoriels. On remarquera que la somme + de deux listes est une méthode pré-définie pour les listes qui consiste à concatener deux listes. Or la somme de deux vecteurs est une méthode différente, on modifiera la méthode + pour un objet vecteur.

Testez votre classe pour faire des opérations entre vecteur.

[ ]:

4 Exercice : Impédance complexe d'un dipôle électrique

L'objectif est de calculer à l'aide de python l'impédance équivalente de n'importe quel dipôle consistué d'une association de résistance, condensateur et bobine.

Créez des classes dipôle, résistance, bobine, ... et les méthodes de tracer des impédances complexes et des lois d'association série et parallèle. On cherchera à ce qu'un utilisateur puisse rapidement calculer et afficher l'impédance complexe du dipôle équivalent d'un circuit.

[ ]:

5 Interfaces graphiques

Nous allons dans cette partie reconnaître le caractère orienté objet dans la réalisation d'interfaces graphiques à l'aide de la bibliothèque tkinter.

[ ]:
```
from tkinter import *

fenetre = Tk()

label = Label(fenetre, text="Hello World")
label.pack()

fenetre.mainloop()
```

Executer les lignes de commandes ci-dessus et en déduire quels sont les objets, attributs et méthodes utilisés.

La bibliothèque tkinter est très utiles pour créer des interfaces graphiques car elle possède de nombreux widgets. Un widget est un élément graphique que l'on vient rajouter à la fenêtre de notre interface utilisateur.

On reconnaîtra pour chaque widgets une structure de la forme
un_widget = UnWidget(widget_parent, un_parametre='une_valeur')

On a déjà utilisé un widget celui d'un Label pour écrire le message Hello World.

On peut voir d'autre widget qui permettent à l'utilisateur d'interagir avec la fenêtre comme par exemple par l'utilisation d'un bouton.

[ ]:
```
fenetre = Tk()

bouton=Button(fenetre, text="Fermer", command=fenetre.quit)
bouton.pack()

fenetre.mainloop()
```

Il y a aussi plein d'autres éléments graphiques pour permettre à l'utilisateur d'intéragir avec l'interface graphique, à vous de choisir.

[ ]:
```
# checkbutton
bouton = Checkbutton(fenetre, text="Nouveau?")
bouton.pack()

# entrée
value = StringVar()
value.set("texte par défaut")
entree = Entry(fenetre, textvariable=string, width=30)
entree.pack()

# radiobutton
value = StringVar()
bouton1 = Radiobutton(fenetre, text="Oui", variable=value, value=1)
bouton2 = Radiobutton(fenetre, text="Non", variable=value, value=2)
bouton3 = Radiobutton(fenetre, text="Peu être", variable=value, value=3)
bouton1.pack()
bouton2.pack()
bouton3.pack()
```

Vous pouvez aussi vouloir en placer plusieurs à la fois soit sans rien préciser et ils seront placé automatiquement, soit en utilisant la méthode .grid(column = 0, row = 0) qui place les widgets sur des lignes et colonnes numérotés.

Par exemple le programme ci-dessous place le message "Hello" "World" à la suite au à la ligne.

[ ]:
```
fenetre = Tk()

label1 = Label(fenetre, text="Hello")
label1.grid(column=0, row=0)
label2 = Label(fenetre, text="World")
label2.grid(column=1, row=0)

fenetre.mainloop()
```

[ ]:
```
fenetre = Tk()

label1 = Label(fenetre, text="Hello")
label1.grid(column=0, row=0)
```

```
label2 = Label(fenetre, text="World")
label2.grid(column=0, row=1)

fenetre.mainloop()
```

En explorant la documentation de tkinter vous pouvez explorer toutes les possibilités de réalisation. Ci-dessous deux exemples:

- une fenêtre de calculatrice.
- une application pour montrer le code couleur d'une résistance à partir de sa valeur.

On pourra essayer de reconnaître le fonctionnement de chaque application.
Puis on pourra modifier la calculatrice pour essayer de faire une calculatrice avec des boutons.

[ ]:
```
from math import *

def evaluer(event) :
    chaine.configure(text = '=> ' + str(eval(entree.get())))

# Programme principal ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
fenetre = Tk()
entree = Entry(fenetre)
entree.bind('<Return>', evaluer)
chaine = Label(fenetre)
entree.pack()
chaine.pack()

fenetre.mainloop()
```

[ ]:
```
class Application:
    def __init__(self):
        """Constructeur de la fenêtre principale"""
        self.root =Tk()
        self.root.title('Code des couleurs')
        self.dessineResistance()
        Label(self.root,
              text ="Entrez la valeur de la résistance, en ohms :").grid(row =2)
        Button(self.root, text ='Montrer',
               command =self.changeCouleurs).grid(row =3, sticky = W)
        Button(self.root, text ='Quitter',
               command =self.root.quit).grid(row =3, sticky = E)
        self.entree = Entry(self.root, width =14)
        self.entree.grid(row =3)
        # Code des couleurs pour les valeurs de zéro à neuf :
        self.cc = ['black','brown','red','orange','yellow',
                   'green','blue','purple','grey','white']
        self.root.mainloop()

    def dessineResistance(self):
```

```
        """Canevas avec un modèle de résistance à trois lignes colorées"""
        self.can = Canvas(self.root, width=250, height =100, bg ='ivory')
        self.can.grid(row =1, pady =5, padx =5)
        self.can.create_line(10, 50, 240, 50, width =5)        # fils
        self.can.create_rectangle(65, 30, 185, 70, fill ='light grey', width =2)
        # Dessin des trois lignes colorées (noires au départ) :
        self.ligne =[]           # on mémorisera les trois lignes dans 1 liste
        for x in range(85,150,24):
            self.ligne.append(self.can.create_rectangle(x, 30, x+12, 70,␣
↪fill='black', width=0))

    def changeCouleurs(self):
        """Affichage des couleurs correspondant à la valeur entrée"""
        self.v1ch = self.entree.get()     # la méthode get() renvoie une chaîne
        try:
            v = float(self.v1ch)          # conversion en valeur numérique
        except:
            err =1                        # erreur : entrée non numérique
        else:
            err =0
        if err ==1 or v < 10 or v > 1e11 :
            self.signaleErreur()          # entrée incorrecte ou hors limites
        else:
            li =[0]*3                     # liste des 3 codes à afficher
            logv = int(log10(v))          # partie entière du logarithme
            ordgr = 10**logv              # ordre de grandeur
            # extraction du premier chiffre significatif :
            li[0] = int(v/ordgr)          # partie entière
            decim = v/ordgr - li[0]       # partie décimale
            # extraction du second chiffre significatif :
            li[1] = int(decim*10 +.5)     # +.5 pour arrondir correctement
            # nombre de zéros à accoler aux 2 chiffres significatifs :
            li[2] = logv -1
            # Coloration des 3 lignes :
            for n in range(3):
                self.can.itemconfigure(self.ligne[n], fill =self.cc[li[n]])

    def signaleErreur(self):
        self.entree.configure(bg ='red')  # colorer le fond du champ
        self.root.after(1000, self.videEntree) # après 1 seconde, effacer

    def videEntree(self):
        self.entree.configure(bg ='white')     # rétablir le fond blanc
        self.entree.delete(0, len(self.v1ch))  # enlever les car. présents

# Programme principal :
f = Application()        # instanciation de l'objet application
```

[ ]: