

Programmation orientee objet et interfaces graphiques

corrige

April 4, 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.

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

[1]

[1]

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

```
[2]: 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]
```

[1]

[1]

Appliquons ce même raisonnement aux lignes de commandes suivantes

```
[3]: 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]
```

[1]

[2]

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

```
[4]: 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]
```

[1]

[2]

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

```
[5]: 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].
```

[1]

[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.

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

[1]

[2]

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

```
[7]: 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]

```

[1]
[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.

[8]:

```

a = 3 # On appelle un objet 3 par a (a = 3)
b = a + 4 # On appelle un objet égal à la valeur de a + 4 = 7 par b (a = 3, et
→ b = 7)
c = a # l'objet a, a aussi pour nom c (c = a = 3, et b = 7)
a = b # on redéfinit le nom a, c'est maintenant aussi le nom de l'objet b, mais
→ plus celui de c (c = 3, a = b = 7)
c = 3.14 # on redéfinit le nom de c, c'est maintenant le nom d'un objet égal à
→ 3.14 (c = 3.14, a = b = 7)

print(a,b,c)

```

7 7 3.14

[9]:

```

a = [2,3,7] # On définit un objet [2, 3, 7] de nom a
b = a # On donne aussi le nom b à l'objet a donc b = a = [2, 3, 7]
print(b[1]) # On affiche l'élément de b, b[1] or b = a donc b[1] = a[1] = 3
a[1] = 4 # On modifie l'élément de a donc b = a = [2, 4, 7]
print(b[1]) # On affiche l'élément de b or b = a donc b[1] = a[1] = 4
a = [5,6,7,8] # On définit un nouvel objet [5, 6, 7, 8] que l'on nomme a, on a
→ donc a = [5, 6, 7, 8] et b = [2, 4, 7]
print(b[1]) # On affiche l'élément de b soit b[1] = 4

```

3
4
4

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

a = [1,2,3,4] # On définit un objet a = [1, 2, 3, 4]
exemple(a) # def exemple(arg): On appelle la fonction exemple en posant arg = a
           →= [1, 2, 3, 4]
           # print(arg) on affiche arg[1] = a[1] donc 2
           # arg[1] = 4 on modifie arg = a = [1, 4, 3, 4]
           # arg = [5,6,7,8] on définit un nouvel objet arg donc arg =
           →[5,6,7,8] et a = [1, 4, 3, 4]
print(a[1]) # on affiche a[1] = 4
```

2
4
4

```
[11]: a = [1,2,34,45] # On définit un objet a = [1,2,34,45]
b = a # On nomme l'objet a aussi b donc b = a = [1,2,34,45]
c = a[1] # On définit l'objet c comme l'élément a[1] donc b = a = [1,2,34,45]
        →et c = 2
a[2] = 1 # On modifie l'élément de a donc b = a = [1,2,1,45] et c = 2
a[1] = 5 # On modifie l'élément de a donc b = a = [1,5,1,45] et c = 2
print(b[2]+c) # On affiche b[2]+c = 1+2 = 3
```

3

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 rapprochent d'objets réels.

Une classe est un groupe d'objets 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'objets que sont les personnes avec :

```
[12]: 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()`

```
[13]: 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'objets `Personne`. Testons quel est le type de l'objet `le_prof`

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

```
[14]: __main__.Personne
```

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 défini 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 grâce à la commande `objet.attribut`, par exemple l'attribut `nom` de `le_prof` est une chaîne de caractère vide.

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

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

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

print(le_prof.nom)
```

Bisognin

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

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

```
[18]: le_prof.prenom = "Rémi"

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

Rémi Bisognin

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

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

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

Rémi Bisognin remi.bisognin@ac-versailles.fr

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'autres arguments si besoin.

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

```
[20]: 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()`

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

Bonjour Rémi Bisognin !

Lorsque les méthodes sont correctement définies dans les classes, elles sont ensuite plus faciles d'utilisation car lorsqu'on veut faire une action avec un objet, on possède une liste de méthodes 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.

```
[22]: class Personne():
        """Classe des personnes"""
        def __init__(self, prenom="", amis=[]):
            self.prenom = prenom
            self.amis = amis

        def ajouter_un_ami(self, other):
            self.amis.append(other)

        def afficher_amis(self):
            for ami in self.amis:
                print(ami.prenom)
```

```
[23]: jean = Personne("jean", [])
        paul = Personne("paul", [])
        sophie = Personne("sophie", [])

        sophie.ajouter_un_ami(jean)
```

```
jean.ajouter_un_ami(paul)

jean.afficher_amis()
```

paul

```
[24]: paul.afficher_amis()
```

```
[25]: sophie.afficher_amis()
```

jean

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."

```
[26]: class Personne():
        """Classe des personnes"""
        def __init__(self, prenom="", amis=[]):
            self.prenom = prenom
            self.amis = amis

        def ajouter_un_ami(self, other):
            self.amis.append(other)

        def afficher_amis(self):
            for ami in self.amis:
                print(ami.prenom)

        def les_amis_de_mes_amis(self):
            for ami in self.amis:
                for ami_d_ami in ami.amis :
                    self.amis.append(ami_d_ami)
```

```
[27]: jean = Personne("jean", [])
paul = Personne("paul", [])
sophie = Personne("sophie", [])

sophie.ajouter_un_ami(jean)
jean.ajouter_un_ami(paul)

sophie.afficher_amis()
```

jean

```
[28]: sophie.les_amis_de_mes_amis()

sophie.afficher_amis()
```

```
jean
paul
```

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.

```
[29]: class Personne():
        def __init__(self, nom="", prenom=""): #Initialisation avec possibilité 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, '!')
```

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

Bonjour Rémi Bisognin !

La méthode `__repr__` sert à 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 :

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

```
<__main__.Personne object at 0x000002110D0A24A8>
```

```
[32]: le_prof
```

```
[32]: <__main__.Personne at 0x2110d0a24a8>
```

On peut vouloir modifier cette fonction pour afficher le nom complet

```
[33]: 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
```

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

Rémi Bisognin


```
[35]: le_prof
```

[35]: Rémi Bisognin

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.

```
[36]: class Personne():
        """Classe des personnes"""
        def __init__(self, prenom="", amis=[]):
            self.prenom = prenom
            self.amis = amis

        def ajouter_un_ami(self, other):
            self.amis.append(other)

        def afficher_amis(self):
            for ami in self.amis:
                print(ami.prenom)

        def les_amis_de_mes_amis(self):
            for ami in self.amis:
                for ami_d_ami in ami.amis :
                    self.amis.append(ami_d_ami)

        def __add__(self, other):
            for ami in other.amis:
                self.amis.append(ami)
            return self
```

```
[37]: jean = Personne("jean", [])
paul = Personne("paul", [])
sophie = Personne("sophie", [])

sophie.ajouter_un_ami(jean)
jean.ajouter_un_ami(paul)

sophie.afficher_amis()
```

jean

```
[38]: sophie = sophie+jean
```

```
sophie.afficher_amis()
```

```
jean  
paul
```

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 nationalité de la personne.

```
[39]: 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, '!')
```

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

```
Hello Mme Guilheneuf !
```

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

```
Bonjour Rémi Bisognin !
```

Cette propriété d'héritage 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 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.

```
[42]: class Vector3(list):
      """Classe"""
      def __init__(self,V = [0,0,0]):
          list.__init__(self,V)
          self.V = V

      def norme(self):
          return (self.V[0]**2+self.V[1]**2+self.V[2]**2)**0.5

      def __add__(self,other):
          A = Vector3()
          for i in range(3):
              A.V[i] = self.V[i]+other.V[i]
          return A

      def __sub__(self,other):
          A = Vector3()
          for i in range(3):
              A.V[i] = self.V[i]-other.V[i]
          return A

      def __repr__(self):
          return self.V.__repr__()

      def __mul__(self,other):
          x = self.V[1]*other.V[2]-self.V[2]*other.V[1]
          y = self.V[2]*other.V[0]-self.V[0]*other.V[2]
          z = self.V[0]*other.V[1]-self.V[1]*other.V[0]
          return Vector3([x, y, z])

      def __or__(self,other):
          return (self.V[0]*other.V[0]+self.V[1]*other.V[1]+self.V[1]*other.V[2])
```

```
[43]: V1 = Vector3([1,1,0])
      V2 = Vector3([0,1,0])
```

```
[44]: V1.norme()
```

```
[44]: 1.4142135623730951
```

```
[45]: V2.norme()
```

```
[45]: 1.0
```

```
[46]: V3 = V1 + V2
      V3
```

```
[46]: [1, 2, 0]
```

```
[47]: V4 = V1 - V2
      V4
```

```
[47]: [1, 0, 0]
```

```
[48]: V5 = V1*V2
      V5
```

```
[48]: [0, 0, 1]
```

```
[49]: s = (V1|V2)
      s
```

```
[49]: 1
```

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 constitué 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.

```
[50]: import numpy as np
      import matplotlib.pyplot as plt

      class dipole():
          def __init__(self,Z = lambda omega: 0):
              self.Z = Z

          def affichage(self,omega_min,omega_max,n):
              omega = np.logspace(omega_min, omega_max, n)
              plt.semilogx(omega, np.real(self.Z(omega)), 'r')
              plt.semilogx(omega, np.imag(self.Z(omega)), 'b')

          def __add__(self,other):
              resultat = dipole(lambda omega: self.Z(omega)+other.Z(omega))
              return resultat

          def __or__(self,other):
              resultat = dipole(lambda omega: self.Z(omega)*other.Z(omega)/(self.
→Z(omega)+other.Z(omega)))
              return resultat

      class resistance(dipole):
```

```

def __init__(self,R=0):
    dipole.__init__(self,lambda omega: R+1j*0*omega)
    self.R = R

class condensateur(dipole):
    def __init__(self,C=0):
        dipole.__init__(self,lambda omega: 1/(1j*C*omega))
        self.C = C

class bobine(dipole):
    def __init__(self,L=0):
        dipole.__init__(self,lambda omega: 1j*L*omega)
        self.L = L

```

```

[51]: R1 = resistance(1)
      R1.R

```

```

[51]: 1

```

```

[52]: R2 = R1+R1
      R2.Z(1)

```

```

[52]: (2+0j)

```

```

[53]: R3 = R1|R1
      R3.Z(1)

```

```

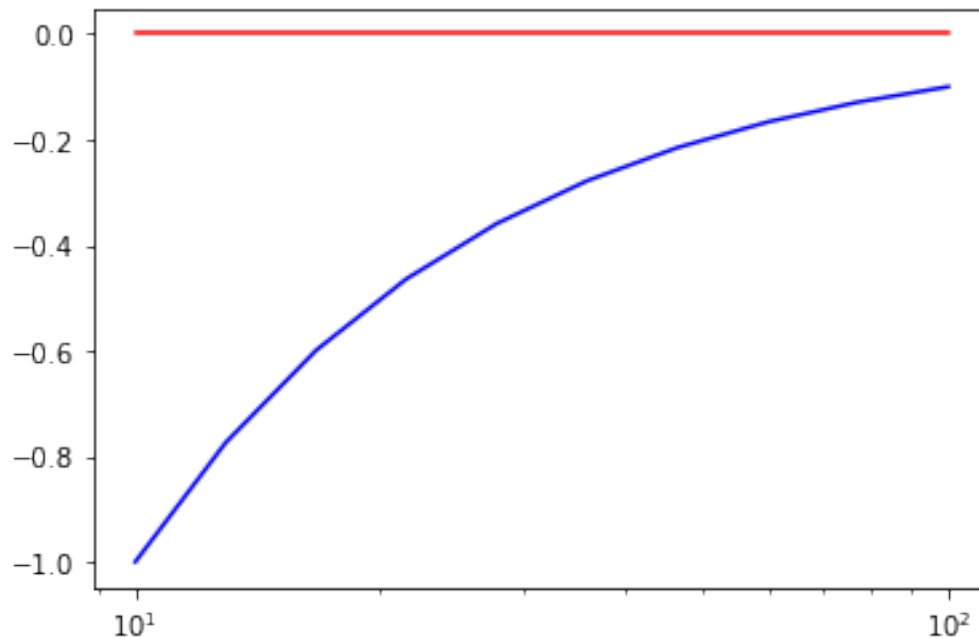
[53]: (0.5+0j)

```

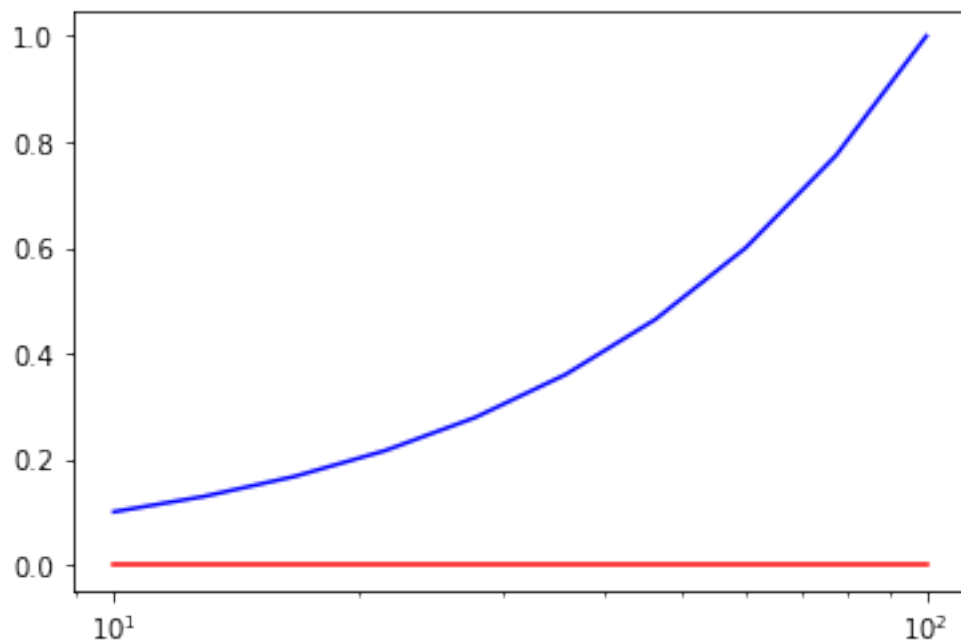
```

[54]: C1 = condensateur(0.1)
      C1.affichage(1,2,10)

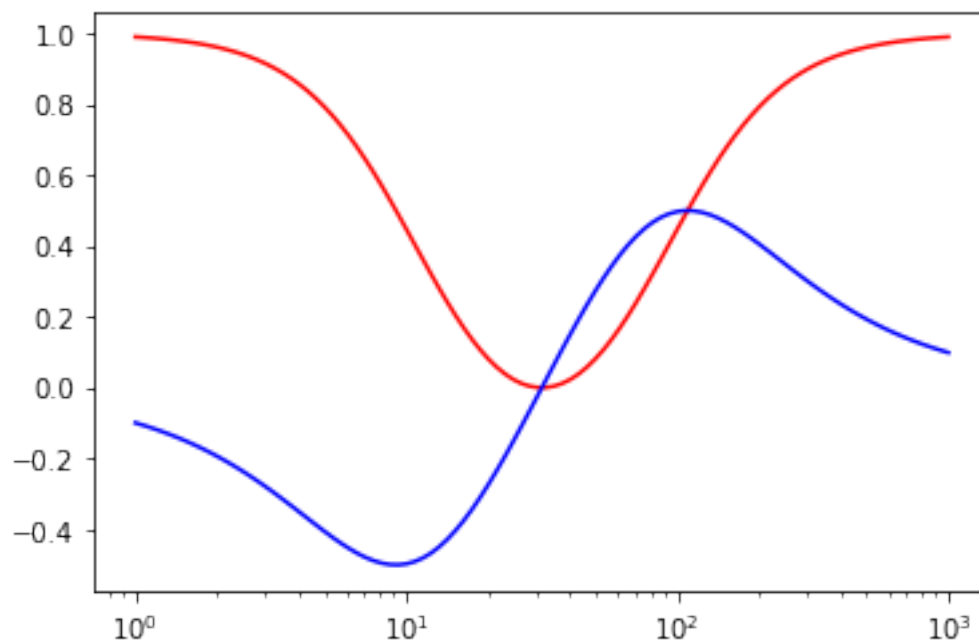
```



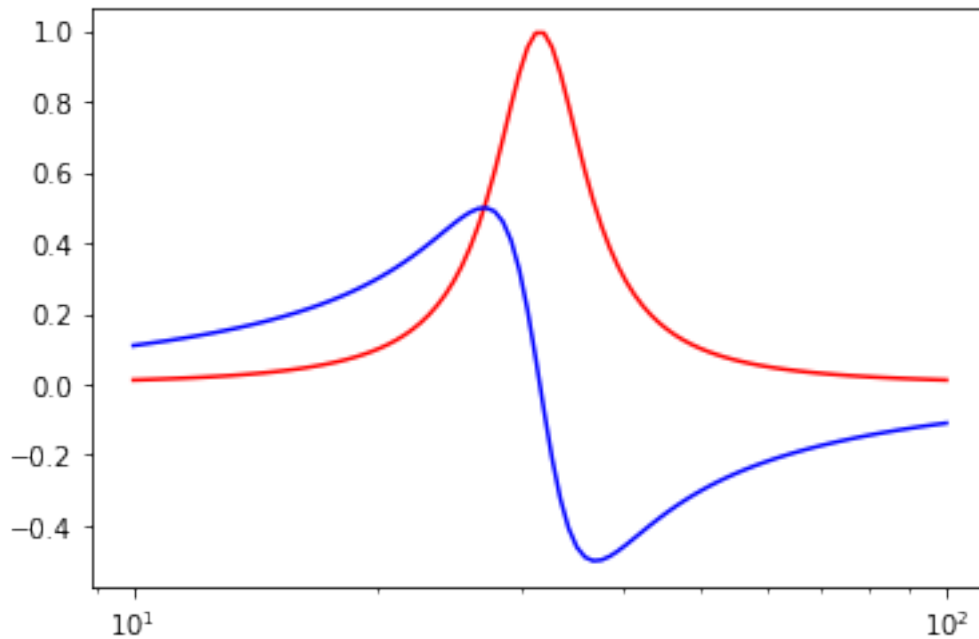
```
[55]: L1 = bobine(0.01)
      L1.affichage(1,2,10)
```



```
[56]: (R1|(L1+C1)).affichage(0,3,100)
```



```
[57]: (R1|(L1|C1)).affichage(1,2,100)
```



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.

```
[58]: 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'autres widget qui permettent à l'utilisateur d'interagir avec la fenêtre comme par exemple par l'utilisation d'un bouton.

```
[59]: 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'interagir 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 placés automatiquement, soit en utilisant la méthode `.grid(column = 0, row = 0)` qui place les widgets sur des lignes et colonnes numérotées.

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

```
[60]: 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()
```

```
[61]: 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.

```
[62]: 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()
```

```
[63]: 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 mémoriserà 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()      # instantiation de l'objet application

```

[: