Programmation orientée objet

DANS CE CHAPITRE on présente les idées de la programmation orientée objet (POO) *via* les exemples des polynômes, des fractions rationnelles puis de la gestion des interfaces graphiques fenêtrées. Python est un langage POO, thème dont l'étude est proposée dans le programme officiel.

La POO date des années 70 et est désormais largement répandue. La plupart des langages utilisés professionnellement – comme Python dont on rappellera que l'un de ses principaux utilisateurs est ... Google – sont *orientés objet*.

§ 1. Introduction

1.1 Principe général d'une classe, objets et méthodes

En première approche, on pourra penser une *classe d'objets* comme un « type de variables » pour lequel on définit certaines opérations (*méthodes*) dédiées. Pensez par exemple au type `list` et aux méthodes : `append, clear, copy, count, extend` ...

Remarque : sous l'interface de Pyzo, taper `liste=[]` puis `liste.` et faire défiler toutes les méthodes pouvant s'y appliquer de même que les attributs privés `__xxx__` En matière de syntaxe, on retiendra aussi que l'application d'une méthode `meth` à un objet `obj` se fait par `obj.meth()`.

Pour le type `string` on connaît également quelques méthodes spécifiques. Ce qui est remarquable – c'est notre premier exemple de *surcharge* d'opérateurs – les instructions `print` et `+` n'ont pas le même effet suivant qu'il s'applique entre deux `string` où il agit comme une mise « bout à bout », on dit une concaténation, ou bien qu'on « ajoute » deux flottants auquel cas il effectue ce que l'on attend de lui. Quelques exemples :

EXERCICE 1
Taper (un copier/coller depuis le PDF par exemple *) les programmes suivants et commenter.

```
p = [1,2,3]
print(p)
len(p)
q = [10,12]
r = p+q
s = p*q
2*p
```

```
s = 'abcde'
print(s)
len(s)
t = 'xyz'
u = s+t
v = s*t
2*s
```

```
A = 3.1415926
print(A)
len(A)
B = 1.5
C = A+B
D = A*B
2*A
```

La proximité avec les structures algébriques en mathématiques et les opérations comme $+$ ou $\times$ dont l'effet diffère suivant les objets (au sens commun) auxquels ils s'appliquent est apparente : $\mathbb{R}$ ou $\mathcal{M}_n(\mathbb{R})$ par exemple. Ceci pour les lois internes, mais que fait donc Python lorsque l'on tape `[0]*10` ? La personne qui a défini la classe *list* a indiqué ce que l'on doit entendre par le « produit » d'une liste par un entier positif : une loi de composition externe comme celle que l'on utilise en mathématiques pour multiplier un vecteur par un scalaire.

Un exemple « concret » : une classe Woiture.

On souhaite définir les objets d'une classe Woiture. Ces objets possèdent des *attributs* que l'on peut penser comme des caractéristiques qui peuvent être de types différents.

```
class Woiture:
    def __init__(self,places=4,coul='jaune',OK=True):
        self.places   = places
        self.couleur  = could
        self.CT       = OK
```

Dans la définition d'une classe on commence toujours par préciser les attributs. C'est ce qui est fait ci-dessus avec la méthode `__init__` : les attributs de l'objet nommé génériquement `self` sont `places,couleur,CT` et ont respectivement pour type `int,string,boolean`. Leurs valeurs par défaut peuvent être indiquées comme ci-dessus : le « véhicule » est jaune, a 4 places et un contrôle technique valide ; par défaut.

*. Peut-être sera-t-il nécessaire alors de modifier les apostrophes

Pour créer un objet de la classe on tape dans la console (par exemple) `totomobile = Woiture()`.

On accède alors à l'attribut `.places` de cette *instance de la classe Woiture – i.e.* de cet objet de cette classe – nommée `totomobile` par `totomobile.places` qui est un entier représentant le nombre de places du véhicule. Et qui vaut ici 4 : la valeur par défaut de cet attribut. On peut changer (que la vie est belle en `Python` ...) la valeur par une simple affectation : `totomobile.places = 10`. On peut aussi décider du nombre de places lors de la définition : `BatMobile = Woiture(places=2,coul='noir',OK=False)`.

Côté *méthodes*, on peut penser aux suivantes : `repeindre` et `demarrer` qui permettent respectivement de changer la couleur par `nom.repeindre(nouvelle_couleur)` et de rouler ... À la suite de la définition de `__init__` on pourra écrire

```
def repeindre(self,newcol):
    self.couleur = newcol
```

Pour la méthode `demarrer` – qui s'appellera donc par `nom.demarrer()` dans la console – il serait bon de ne pas laisser le conducteur rouler sans avoir un contrôle technique valide. Il faudra donc gérer une potentielle « erreur », on parle d'`Exception`, correspondant au cas `nom.CT==False`. Cela est possible en `Python`, on verra sa gestion précise lors du paragraphe suivant avec le cas de l'exception `PileVide` empêchant de dépiler une pile vide.

1.2 Un exemple simple : une classe Pile

Comme on l'a dit avant, une *classe* est une forme, plus évoluée, de type de variables comme `int`, `float`, `char`, `string` : il définit une famille d'objets avec certains *attributs* et des *méthodes* propres comme la façon d'afficher l'objet *via* `print` ou bien d'additionner par +.

Une *exception* est une *instance* de la classe existante `Exception` qui permet de gérer les « erreurs » qui peuvent survenir lors de l'execution d'un programme : par exemple une division par zéro. Déclarer une exception indique au programme comment réagir quand l'« erreur » associée à l'exception se produit : au lieu de s'arrêter brutalement, le programme applique la marche à suivre décrite dans la définition de l'exception et peut, éventuellement, continuer son exécution.
Cette situation est tout à fait adaptée à ce dont on a besoin pour traiter des piles vides.

```
class PileVide(Exception):
    def __init__(self):
        self.argh = 'On ne dépile pas une pile vide'
    def __str__(self):
        return self.argh
```

- Ci-dessus on déclare une *sous-classe* nommée `PileVide` de la classe existante `Exception`.
- Ensuite il y a le passage obligatoire de l'initialisation où l'on définit les attributs de `self` qui désigne un objet de la classe `Pilevide` que nous sommes en train de définir. Ici, il y a un unique attribut à `self` nommé `argh` qui est une chaîne de caractère.
- Enfin on définit la méthode `__str__` qui décrit la façon d'afficher par `print` le contenu d'un objet de la classe. Ici le simple `return self.argh` revient à faire afficher `self.argh` *i.e.* la chaîne `'on ne dépile pas une pile vide'` lorsque l'on tentera de dépiler une pile vide.

Le temps est venu de notre première vraie déclaration de classe.
En voici la définition complète (on parle de *constructeur*) que l'on explique ensuite point par point.

```
class Pile:
    # on commence par les attributs
    def __init__ (self,donnee=[]):
        self.content = donnee
    # puis les methodes
    def vide(self):
        return (self.content == [])
    def __len__(self):
        return len(self.content)
    def __str__(self):
        if self.vide():
            return 'Pile vide'
        else:
```

```
            chaine = ';'.join(str(c) for c in self.content)
            return 'Dessous :'+chaine
    def empile(self,c):
        self.content = self.content+[c]
    def depile(self):
        if self.vide():
            raise PileVide()
        else:
            c = self.content[-1]
            self.content = self.content[:-1]
            return c
```

1° Par `class Pile:` on déclare une nouvelle classe nommée `Pile` et l'on décrit ensuite les attributs et méthodes des objets de la classe. Une *instance* de `Pile` *i.e.* ce qui sera une « variable de type pile » dans le programme est génériquement notée `self`.

2° Avec `__init__` on dit comment initialiser un objet de la classe. L'attribut optionnel `donnee` est fourni par l'utilisateur ou bien affecté à la valeur par défaut spécifiée ici `[]` pour la pile vide.
On désignera ensuite le contenu de la pile par `self.content` et ce dernier est affecté lors de la création de l'objet à `donnee`.

3° On définit une méthode `vide` qui renverra un booléen suivant l'état de l'objet `nom` de la classe `Pile` et qui sera appelée par `nom.vide()`.

4° On redéfinit les méthodes générales `__str__` pour l'affichage et `len` pour la taille afin de pouvoir toujours les appeler avec `print(nom)` et `len(nom)` dans le programme mais en décrivant l'effet spécifique qu'elles doivent avoir sur un objet de la classe `Pile`.

5° *Remarque :* Les méthodes font partie intégrante du constructeur de la classe : elles sont *encapsulées* dans la définition de la classe. Ce mode de conception incite à la rigueur et à bien penser les variables/objets dont on a besoin pour résoudre un problème.

EXERCICE 2
Après avoir exécuté le script précédent (fichier disponible sur le net), précédé de celui définissant `PileVide` (à disposition dans le répertoire commun ou copier/coller depuis la version PDF de ce document ; attention toutefois de retaper les apostrophes qui ne sont pas bien copiées) :

1° Taper dans la console `test = Pile(); test.vide(); print(test)`.
Puis `test.empile('qqch'); print(test); test.depile(); test.depile()`
Et enfin, `test.empile(2); test.empile(3); print(test); len(test)`.

2° En utilisant exclusivement les méthodes de la classe `Pile`, définir deux objets de la classe `Pile`, nommés `un` et `deux` tels que `un` vaut la liste des carrés des dix 1ers entiers non nuls et `deux` est vide.
Compléter le script de sorte qu'à la fin `un` soit vide et que `deux` contienne la liste des carrés de $20, 18, \ldots, 2$.

EXERCICE 3
Ajouter à la définition de la classe `Pile` une méthode `envers` qui transforme le `content` d'un objet de la classe `Pile` en la liste « inverse » en le sens que l'ordre de ses éléments est inversé.
Testez votre travail dans la console.

EXERCICE 4
Enregistrer le constructeur de la classe `Pile` – cela comportant aussi la définition de l'exception `PileVide` et toutes les méthodes – sous le nom `Repertoire_de_pyzo/lib/python3.4/site-packages/pile.py`
Dans une nouvelle session de Pyzo, taper dans la console `from pile import *`.
On dispose désormais d'un nouveau « type » : `Pile` et des méthodes afférentes.
Vérifiez cette affirmation en définissant dans la console une pile `test` et en testant les méthodes `.vide`, `.empile()`, `.depile` par divers affichages *via* `print`.

En résumé : Une classe d'objets est l'équivalent d'un nouveau type de variables qui possède ses méthodes propres. On peut de plus redéfinir des méthodes pré-existantes pour tous les types comme l'affichage, la taille ou l'évaluation pour leur donner un sens/une action particulier(e) adaptée aux objets manipulés.

§ 2. Un exemple plus fouillé : La classe des polynômes

2.1 Premières définitions

1° Définir une classe `Polynome` dont les objets « seront » des polynômes de $\mathbb{R}[X]$ repérés par la liste de leurs coefficients où, de gauche à droite, on lit les coefficients des monômes par ordre croissant de degré.

·) Par défaut il s'agira du polynôme nul.

·) Un attribut `.coefs` sera donc défini valant cette liste.

Par exemple : `P=Polynome([0,3,0,4])` définira un objet de la classe `Polynome` avec `P.coefs` valant la liste `[0,3,0,4]` tout ceci représentant le polynôme $3X + 4X^3$.

2° Écrire une méthode `deg` calculant le degré pour lequel on posera -1 comme valeur pour le polynôme nul.
Indication : on pourra utiliser les méthodes `enumerate` et `reversed` concernant les listes.

3° Écrire les méthodes existantes gérant + et * : `__add__` et `__mul__`.

4° De même pour celles gérant – : `__neg__` (pour l'opposé) et *en déduire* celle de `__sub__`.

Jusqu'à présent il n'est pas très commode de vérifier notre définition en cours de la classe `Polynome` *car nous ne savons pas afficher un objet de cette classe correctement :* `P.coefs` *donne la liste des coefficients ce qui n'est pas très satisfaisant.*
On indique à `Python` *comment afficher un objet de la classe* `Polynome` *en redéfinissant la méthode* `__str__`*, en charge de l'affichage d'un objet via* `print` *et valable pour toutes les classes, mais en explicitant son action pour un objet de la classe* `polynôme`*.*
La (ré)écriture d'une méthode existante est appelée ***surcharge de la méthode****.*

5° Surcharge de la méthode `__str__` pour l'affichage « standard » des polynômes :

a) Commencer par écrire une méthode `str_monome(self,c,i)` qui renvoie la chaîne `'cX^i'`.
Améliorer votre script pour que l'on affiche uniquement `'X^i'` quand `c=1` et `'-X^i'` quand `c=-1` mais aussi que l'exposant ne soit pas affiché lorsqu'il vaut 1 et que `X^0` soit remplacé par `1`.

b) Écrire alors la surcharge de `__str__`.
N.B. : On prendra soin de ne pas afficher le premier `'+'` le cas échéant.

c) Testez largement dans la console par exemple avec

```
P=Polynome([0,0,1]) # puis avec [-1,1,2,-3] et [0] et ...  print(P)
```

Afin de vérifier que la méthode d'affichage fait bien ce qu'on lui demande dans tous les cas.

6° Testez divers affichages afin de vérifier que les méthodes écrites précédemment fonctionnent bien.
En particulier : S'assurer de la bonne la gestion du premier signe et celui du terme constant.

2.2 D'autres méthodes à écrire

7° `affiche` : écrit à l'écran un polynôme mais par ordre décroissant de puissance de ses monômes.

8° `derive` et `integre` : renvoient respectivement la dérivée d'un polynôme et sa primitive s'annulant en 0.

9° Surcharger la méthode existante `__call__` qui permet d'évaluer un polynôme `P` par `P(-3)` par exemple.

a) Premièrement : classique en calculant la valeur de chaque monôme puis en sommant.

b) Deuxièmement : par la *méthode de Hörner* qui repose sur l'identité

$$P(x_0) = p_0 + x_0(p_1 + x_0(p_2 + x_0(p_3 + x_0(\cdots + x_0(p_n))))).$$

Le faire en deux versions : l'une itérative et l'autre récursive.
Indication : Dans ce dernier cas on écrira quelque chose comme `x*Polynome([???]).__call__(x)`.

c) Comparer les « coûts » de ces deux approches avec les nombres de multiplications et d'additions.

10° `div_eucl(self,other)` : retourne une liste de deux polynômes ; quotient et reste de la division euclidienne de `self` par `other`. *Remarque :* c'est ainsi, par `self` et `other` – ou `toto` – que l'on indique deux arguments pour une méthode qui appartiennent tous deux à la classe en cours de définition.

11° Un peu d'analyse numérique :

a) `integrale(self,a,b)` : valeur « exacte » *via* `self.integre()` de $\int_a^b$ `self`.

b) `riemann(self,a,b,n)` : valeur approchée de $\int_a^b$ `self` par la méthode des rectangles avec `n` itérations.

c) `dicho(self,a,b,eps)` : valeur approchée à `eps>0` près de, la supposée existante et unique, racine de `self` sur `[a,b]` par la méthode de dichotomie (un *must* !).

d) `newton(self,x0,n)` : valeur approchée d'une racine de `self` par la méthode de Newton (on suppose que les hypothèses sont vérifiées ...) à partir de `x0` et pour `n` itérations.
Indication : On pourra affecter `f = self` et `fp = f.derive()`

2.3 Classes dérivées : les Fractions rationnelles

2.3.1 Héritage (simple)

La classe polynôme étant définie, il est possible de définir une nouvelle classe dite *dérivée* de la classe polynôme qui *héritera* des méthodes de sa *classe mère* mais dans laquelle certaines méthodes existantes (penser au degré par exemple) pourront être redéfinies. La syntaxe est

```
Class ClasseFille_derivee(ClasseMere_parente)
```

Les objets d'une classe dérivée *héritent* des méthodes de la classe *parente*; c'est la notion d'*héritage*.

EXERCICE 5
Construire une classe dérivée de la classe `Polynome` appelée `FracRat` et y redéfinir les méthodes `deg`, `__str__`, `__add__`, `__neg__`, `__sub__`, `__mul__` et `__div__`.

2.3.2 Héritage multiple (ici double)

On va définir deux classes : la première est la classe des rationnels et la seconde des fractions rationnelles qui sera *dérivée* à la fois de la classe des rationnels et de celle des polynômes.

Les objets d'une classe dérivée de plusieurs classes mères *héritent* des méthodes de chacune de leurs classes *parentes*; c'est la notion d'*héritage multiple ou de polymorphisme*.

Remarque : Des questions épineuses et licites arrivent de suite comme de savoir de quelle méthode hérite un objet de `Fille(mere1,mere2)` lorsque cette méthode est différente pour `mere1` et `mere2` ...

EXERCICE 6
Définir une classe `Rationnel` et les méthodes `__str__`, `__add__`, `__neg__`, `__sub__`, `__mul__` et `__div__`.

Voici alors une définition de classe `FracRat` comme dérivée de deux classes mères :

```
class FracRat(Polynome,Rationnel):
    def __init__(self,num,denom):
        self.num=num
        self.denom=denom

    def deg(self):
        # utilise le deg des polynomes
        return (self.num).deg()-(self.denom).deg()

    def __str__(self):
        return Rationnel(self.num,self.denom).__str__()

    def __add__(self,other):
        n = self.num*other.denom+self.denom*other.num
        d = self.denom*other.denom
        return Rationnel(n,d)
```

EXERCICE 7
Compléter la définition de la classe dérivée ci-dessus en écrivant les méthodes `__neg__`, `__sub__`, `__mul__` et `__div__`. Ces méthodes se substituent alors à celles de la (des) classe(s) mère(s).

EXERCICE 8 (AFFINAGE DE LA CLASSE RATIONNEL *per se*)
Voici quelques améliorations de la classe `Rationnel` lorsque vue comme une classe dédiée aux entiers et non plus comme classe parente d'une classe pour les fractions rationnelles.

1° Créer une exception pour la division par zéro et une méthode `simplif(self)` afin de simplifier et remplacer les attributs `num` et `denom` de `self` par les numérateurs et dénominateurs de la fraction irréductible correspondante.
Cette méthode en utilisera une autre, appelée `pgcd(self,a,b)` et encapsulée elle aussi dans le constructeur de `Rationnel`, qui renverra le PDCG de `a` et `b` par la méthode récursive des soustractions successives.

2° Ajouter une méthode `estEntier` qui renvoie un booléen suivant que le rationnel est ou non un entier relatif et une méthode `__float__` qui renvoie une valeur approchée de type flottant.

3° Les anglo-saxons aiment à écrire les nombres rationnels sous la forme $\lfloor x \rfloor + \frac{a}{b}$ où $0 \leqslant \frac{a}{b} < 1$.
Ajouter à la classe `Rationnel` une méthode `display(self)` qui affiche à « l'anglaise » le rationnel `self`.

§ 3. Un exemple d'application à la gestion des fenêtres

La programmation orientée objet est particulièrement adaptée à la gestion des interfaces graphiques. Les fenêtres sont des objets ayant plusieurs attributs comme des boutons, des textes, des zones de saisies qui interagissent et ont, en quelque sorte, une « vie propre ».
Dans la programmation séquentielle classique, les instructions sont exécutées linéairement de « haut en bas ». Comment programmer, dans ce cadre, la gestion de plusieurs fenêtres dont le comportement dépend des mouvements de souris, des actions de l'utilisateur ? La chose n'est pas aisée et de toute façon très peu naturelle.
Au contraire, en programmation orientée objet, *une fois une classe adaptée définie*, on dispose d'objets comme une fenêtre, un bouton, un menu déroulant ... possédant tous des attributs adéquats qui permettent leur « vie autonome ». On s'initie après à la gestion de l'interface graphique avec la librairie `Tkinter`.

3.1 La librairie `Tkinter`

Cette bibliothèque permet de gérer fenêtres, boutons, ascenseurs et surtout la façon dont ils réagissent à divers *événements* comme un clic de souris, l'entrée d'un texte, la destruction d'une fenêtre ...

Avec `pyzo` sous Mac tout d'abord changer dans les « préférences » le Gui pour Tk puis relancer

Tout commencera bien sûr avec `from tkinter import *`
On aura tout de suite besoin de

- `fenetre = Tk()` : créé la fenêtre principale (dans laquelle tout se passera), son nom (pour accéder à ses attributs) est `fenetre` et peut être changé, d'autres créées ...
- Pour créer un objet `txt` contenant un texte au sein de `fenetre` :
  `txt = Label(fenetre,text='le texte voulu')`
- ⚡ La ligne précédente n'affiche pas le texte, elle le définit juste, pour le placer et l'afficher on utilise la méthode `pack` en executant `txt.pack()`
- La fenêtre est définie mais n'a pas encore d'existence concrète.
  On lui « donne vie » en tapant `fenetre.mainloop()` qui s'achève lorsque l'on ferme la fenêtre.
- *Remarques :*
  - La fenêtre accepte « déjà » (merci `Tkinter`) un redimensionnement à la souris, un déplacement, et bien sûr sa destruction. Testez !
  - Une fois « lancée » la méthode `mainloop`, la fenêtre « s'autogère » si l'on peut dire et le programmeur a les mains libres pour s'occuper de la suite de son algorithme. Cela est vraiment rendu possible par l'orientation objet du langage et par sa librairie d'interfaçage graphique pour la gestion des fenêtres.
  - Les éléments de la fenêtre – pour laquelle on parle aussi d'*application* – sont des *widgets* pour la contraction de *window gadgets* (textes, boutons, ascenseurs, ...). Certains peuvent être modifiés après leur création †. Essayez par exemple de taper dans la console alors que `fenetre` est active :
    `txt["text"]='Au revoir'`

Voici d'autres fonctionnalités basiques :

- **Déclaration d'un bouton**.
  `bouton = Button(fenetre, text = 'quitter',command = fenetre.destroy)`
  - Un bouton est créé, comportant en son centre le texte 'quitter' et qui ferme la fenêtre lorsque l'on clique dessus : méthode `destroy`.
  - On peut définir soi même l'action à faire lors du clic sur un bouton appelé `press`. Par exemple, pour une action nommée `agir` et qui affiche `'toto'` on pourra écrire

```
def agir():
    print('toto')
press = Button(fenetre, text = 'Pressez', command = agir)
press.pack()
```

  - Il convient de placer et afficher ce bouton avec la méthode déjà vue `pack` :
    `bouton.pack()`
    Noter que cela peut se faire dans la console, un peu comme le changement d'un texte de `Label` comme avant. Testez alors ce que produit un clic sur le bouton.

†. Ne semble pas fonctionner sous windows car la console n'est plus accessible lorsqu'une `mainloop` est active ...

·) **Les widgets `Entry`**

Pour pourvoir « entrer » des données dans un champ d'une fenêtre ; un peu comme avec `input`

```
var_texte = StringVar()
ligne_texte = Entry(fenetre, textvariable=var_texte, width=30)
ligne_texte.pack()
```

`var_texte` est le nom de la variable dans laquelle sera stocké le texte entré par l'utilisateur et sa longueur (`width`) est de 30 caractères.

Notez qu'il peut être utile d'indiquer, comme pour un `input`, ce que l'on attend de l'utilisateur par l'ajout d'un texte accompagnant la *zone de saisie* `Entry`. On peut par exemple utiliser un widget `Label`.

3.2 Gestion des événements et autres widgets

Voici deux questions basiques au sujet de la gestion des widgets `Entry` et `Button` :

·) Comment récupérer, par exemple après validation sur la touche « Entrée », le texte entré par l'utilisateur ?

On doit définir une fonction qui réagit à l'*événement* `'<Return>'` qui signifie que l'utilisateur à validé sa saisie par « Entrée ».

On « lie » `bind` en anglais le widget `Entry` à une action précise dans ce cas et l'on utilise la méthode `get` pour récupérer le contenu de la zone de saisie.

```
def valide(event):
    print(entree.get())

entree = Entry(fenetre, width = 50)
entree.bind('<Return>',valide)
entree.pack()
```

·) EXERCICE : Comment compter le nombre de clics sur un bouton ?

On utilise une astuce en créant un `Label` qui contient le nombre de clics courant. Si l'on veut juste en faire le compte sans l'afficher il suffit de ne pas faire appel à la méthode `pack` pour ce `Label` auxiliaire.

```
def ajoute():
    cmpt["text"]=str(int(cmpt["text"])+1)

cmpt=Label(fenetre,text='0')
cmpt.pack()
compte = Button(fenetre,text='Clic',command=ajoute)
compte.pack()
```

D'autres `widgets`

1° `Frame` : définit un cadre dont on peut choisir les dimensions, la couleur de fond ...

```
cadre=Frame(fenetre,width=200,height=100,bg='light-yellow',borderwidth=1)
cadreTitre=LabelFrame(cadre,text='titre du cadre')
```

2° `RadioButton` : un bouton multiple, un seul choix possible

```
choix = StringVar()
choixrouge = Radiobutton(fenetre, text="Rouge", variable=choix, value="rouge")
choixvert = Radiobutton(fenetre, text="Vert", variable=choix, value="vert")
choixbleu = Radiobutton(fenetre, text="Bleu", variable=choix, value="bleu")
choixrouge.pack()
choixvert.pack()
choixbleu.pack()
```

L'important est que ces, *a priori*, différents `RadioButton(s)` se réfèrent tous à la même variable ici `choix` qui contient le numéro de l'option sélectionnée.

Exécuter le script précédent pour visualiser son effet.

3° `Canvas` : les canevas pour afficher des dessins, des images dans un cadre.

Voici un exemple sophistiqué, l'exécuter ‡ et en comprendre la syntaxe en observant l'effet produit.

‡. Fichier à disposition sur internet sous le doux nom de `canvas.py`

```
from random import randrange

# --- définition des fonctions gestionnaires d'événements : ---
def drawline():
    '''Tracé d'une ligne dans le canevas can1'''
    global x1, y1, x2, y2, coul
    can1.create\_line(x1,y1,x2,y2,width=2,fill=coul)
    # modification des coordonnées pour la ligne suivante :
    y2, y1 = y2+10, y1-10

def changecolor():
    '''Changement aléatoire de la couleur du tracé'''
    global coul
    pal=['purple','cyan','maroon','green','red','blue','orange','yellow']
    c = randrange(8)            \# => génère un nombre aléatoire de 0 à 7
    coul = pal[c]

#------ Programme principal -------

# les variables suivantes seront utilisées de manière globale :
x1, y1, x2, y2 = 10, 190, 190, 10            # coordonnées de la ligne
coul = 'dark green'                          # couleur de la ligne

# Création du widget principal ("maître") :
fen1 = Tk()
# création des widgets "esclaves" :
can1 = Canvas(fen1,bg='dark grey',height=200,width=200)
can1.pack(side=LEFT)
bou1 = Button(fen1,text='Quitter',command=fen1.destroy)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1,text='Tracer une ligne',command=drawline)
bou2.pack()
bou3 = Button(fen1,text='Autre couleur',command=changecolor)
bou3.pack()

fen1.mainloop()             # démarrage du réceptionnaire d'événements
```

3.3 Quelques exercices

1° Créer deux fenêtres, l'une contenant un widget `Entry` et l'autre un widget `Label` qui affiche la température saisie dans la première en degrés Celcius lors la validation par « Entrée » dans la seconde convertie en Farenheit.

2° Créer une application (fenêtre) avec un `Entry` dans lequel l'utilisateur entrera un calcul à faire effectuer et un `Label` dans lequel le résultat sera affiché après appui sur « Entrée ».
Conseil : importer la librairie `math` si vous voulez faire effectuer des calculs avec des fonctions plus compliquées que les 4 opérations et utiliser l'instruction `eval(chaine)` qui retourne le résultat de type flottant de l'opération contenue dans la string `chaine`.
Par exemple : `eval('5*2-1')` renvoie le résultat numérique 9.

3° Sachant que `'<Button-1>'` est l'événement « clic gauche », et que pour un tel `event`, les méthodes `x` et `y` fournissent les coordonnées du pointeur de la souris, créer une application qui affiche les coordonnées du pointeur à chaque clic gauche du mulot dans le cadre de la fenêtre de l'application.

4° Dans un `Canvas` de taille 400×300, faire afficher des disques rouges de rayon 10 pixels centrés en chaque clic gauche de la souris. On ajoutera un bouton « Quitter » qui fermera alors la fenêtre.
Conseil : L'instruction `can.create_oval(x-r,y-r,x+r,y+r,width=2,fill='red')`
affiche dans le `Canvas` nommé `can` un disque rouge de centre (x, y) et de rayon r.

5° Créér une application comprenant 3 widgets : un bouton pour quitter qui sera situé en bas de la fenêtre, un `Radiobutton` comprenant deux options « Ovale » et « Rectangle » et un `Canvas` nommé `cadre`.

a) Lorsque l'on clique avec le bouton gauche, on sauve les coordonnées du pointeur de la souris dans deux *variables globales* `x1` et `y1`, lorsque l'on clique bouton droit (à condition que l'on ait avant cliqué bouton gauche : pour gérer cette fonctionnalité on pourra utiliser une variable booléenne), alors on trace un « Ovale » ou un « Rectangle » suivant l'option cochée pour le `Radiobutton`.

On utilisera et complètera :

```
def o():
    # lors de la selection ovale on affecte 1 a la variable forme
    forme.set(1)

forme = IntVar()
ovale = Radiobutton(cadre, variable=forme,text='Ovale',value=1, command=o)
```

b) Étoffer ce « logiciel » de dessin avec d'autres formes : « droite », « segment », « cercle » (un clic gauche pour sélectionner le centre et un clic droit pour un point du cercle).

c) Autre option possible : ajuster un `Radiobutton` pour choisir la couleur.

6° Un autre logiciel de dessin.

a) Cette application contient un `canvas` et trois `Checkbutton`
   - Un interrupteur « ON / OFF » signalant si le stylo est posé ou non ;
   - un autre *toggle* pour la symétrie d'axe horizontal au centre du canevas ;
   - un dernier *toggle* pour la symétrie d'axe vertical au centre du canevas ;

b) Lorsque le stylo est posé, alors on dessine sur le canevas et on trace en même temps les symétriques si les `Checkbutton` sont cochés.

7° Un traceur de courbes paramètrées :

a) L'application contient 5 widgets `Entry` dont les textes sont `x(t),y(t),tmin,tmax,nbe` mais aussi un `Button` de texte « Dessiner », et enfin un `Canvas` de taille à choisir.

b) Lors de l'appui sur le bouton « Dessiner », le programme trace la courbe paramétrée correspondant.
   Remarques :
   - On aura besoin des librairies `numpy` (afin de disposer des fonctions mathématiques usuelles) et `matplotlib.pyplot` pour tracer la courbe dans une fenêtre gérée, elle, par cette librairie.

     ```
     from numpy import *
     import matplotlib.pyplot as pp
     ```

   - On aura aussi recours à l'instruction `eval(chaine)` qui « évalue » l'expression mathématique codée dans la chaîne de caractères `chaine`.
     Par exemple : `eval('cos(pi/3)')` retourne `0.5`.
   - Pour le tracé : après avoir calculé toutes les valeurs du paramètre pour lesquelles on veut calculer les coordonnées du point de la courbe par
     `T = linspace(eval(tmin.get()),eval(tmax.get()),int(eval(nbepts.get())))`
     On pourra alors calculer les abscisses par le très « pythonesque »
     `X = [eval(x.get()) for t in T]`
     Puis on tracera par `pp.plot(X,Y)` suivi de `pp.show()`.
   - On pourra essayer la courbe $x(t) = t\sin(t)$ et $y(t) = t\cos(2t)$ sur $[-\pi, \pi]$ avec 100 points.

c) Ajouter un `Checkbutton` « Quadrillage » qui affiche un quadrillage.
   De même pour forcer un repère orthonormé.
   Voir les instructions `pp.grid(True)` et `pp.axis('equal')`.

d) On pourra aussi faire tracer les axes de coordonnées, chercher l'instruction adéquate.

e) On pourra ajouter un `Radiobutton` proposant plusieurs couleurs différentes pour la courbe.

f) Et même avec `Sympy` et `numpy` il devrait être possible de faire déterminer les points stationnaires, les équations de tangentes en des points réguliers, voire de calculer les variations de x et y ...