CM4: type *class* et interfaces graphiques

Programmation 2: L2 mathématique

Florian Bridoux

November 15, 2019

Le type class

Le type *class* permet de faire de la "programmation orientée objet", avec des "objets" qui possèdent à la fois des données (on parle "d'attributs") et des fonctions pour les manipuler (on parle de "méthodes"). Exemple de déclaration d'une classe :

```
class Point :
    def __init__(self, px, py) :
        print("Initialisation d'un point")
        self.x = px
        self.y = py
    def afficher(self) :
        print (self.x, self.y)
```

On a déclaré une classe Point, c'est un nouveau type :

```
>>> type(Point)
<class 'type'>
```

Le type class

On met toujours une Majuscule aux noms de classes.

1. les fonctions __init__ et afficher sont ses "méthodes" ;
2. le paramètre self est passé en premier à chaque méthode, il désigne la variable courante de type Point ;
3. chaque variable de type Point aura ses propres "attributs" x et y

Le type class

Déclaration d'une variable :

```
>>> p = Point()              # Erreur ... missing 'px' and 'py'
>>> p = Point(3,5)           # Initialisation d'un point
>>> p.afficher()             # 3 5
```

Donc,

1. lorsqu'on appelle p = Point(3,5), la méthode __init__ est appelée avec les paramètres p, 3, 5 (plus exactement, le futur p)
2. lorsqu'on appelle p.afficher(), la méthode afficher() est appelée avec le paramètre p.

Le type class

Vocabulaire:

1. Une variable de type class désigne un "objet" en mémoire, qui est un "exemplaire" de la classe : on dit aussi une "instance" de la classe.
2. "instancier une classe" signifie tout simplement : créer une instance de la classe, c'est-à-dire une variable de type de la classe. Par exemple en écrivant : p = Point(3,4) on a "instancié la classe Point".
3. la méthode __init__ s'appelle le "constructeur" de la classe.
4. dans le constructeur de Point, on a mémorisé les paramètres px,py "en attribut", ou encore, "dans des attributs x et y".
5. le premier paramètre self de chaque méthode est "l'instance courante", ou encore "l'instance".

Le type class

Propriété : une instance peut avoir des attributs en plus de ceux prévus dans __init__ :

```
>>> p = Point(3,5)
>>> p.z = 8
>>> p.z                    # 8
```

(On dit encore que l'on peut "surcharger une instance".) Mais où sont stockés ces attributs ? Dans ce dictionnaire :

```
>>> p.__dict__          # {'z': 8, 'y': 5, 'x': 3}
```

On peut donc rajouter aussi des attributs à p en faisant p.__dict__[clé] = valeur :

```
>>> p.__dict__['t'] = 11
>>> p.t                              # 11
```

On peut examiner une classe ou une instance avec dir() :

```
>>> dir(Point)
>>> dir(p)                # il y a en plus x, y, z, t
```

Le type class

Les attributs entourés de doubles "__" (comme __init__) sont réservés, et ont un rôle spécial. Exemple :

```
>>> p.__class__.__name__      # 'Point'
>>> p.__class__.__str__     # méthode appelé par str(p)
```

Tous les types de python sont en fait des classes :

```
>>> type(1)              # <class 'int'>
>>> type(())             # <class 'tuple'>
>>> type([])             # <class 'list'>
>>> type({})             # <class 'dict'>
>>> type("")             # <class 'str'>
```

Mais on ne peut pas leur rajouter des attributs : ces classes sont "gelées".

Exercice

Créez une classe Matrice. Cette classe contiendra:

- Un constructeur qui prendra deux paramètre x et y et qui initialisera l'instance comme une matrice de dimensions $x \times y$ composé uniquement de 0.
- Une méthode get(self,x,y) qui donnera la valeur courante de la case (x,y) de la matrice courante.
- Une méthode set(self,x,y,val) qui donne la valeur *val* à la case (x,y) de la matrice courante.
- Une méthode afficher(self) qui affiche l'instance courante dans le terminal.

Fenêtre

Importer les modules nécessaires :

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk
```

Création d'une fenêtre principale :

```
win = Gtk.Window()
```

Elle n'apparait pas (encore).

Fenêtre

On lui donne une taille initiale :

```
win.set_size_request(400, 300)
```

On la marque pour qu'elle soit visible ainsi que son contenu:

```
win.show_all()
```

Enfin on appelle la "boucle d'événements GTK":

```
Gtk.main()
```

La fenêtre devient visible.

Fenêtre

Fenêtre

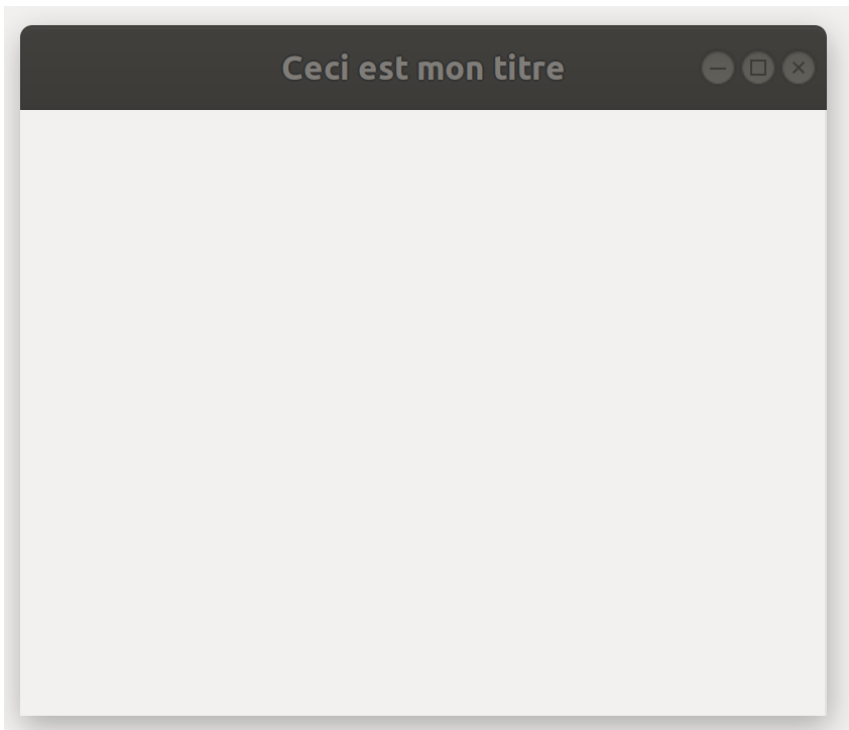
Si on clique sur l'icone de fermeture, la fenêtre disparait mais Gtk.main() ne s'arrête pas. Il faut le lui dire :

```
win.connect("destroy", Gtk.main_quit)
```

"destroy" est un "signal", et lorsqu'il y a lieu, Gtk.main() appelle Gtk.main_quit() qui fait sortir de Gtk.main() et détruit la fenêtre.

Enfin, on peut donner un titre à la fenêtre :

```
win.set_title("Ceci est mon titre")
```



Fenêtre

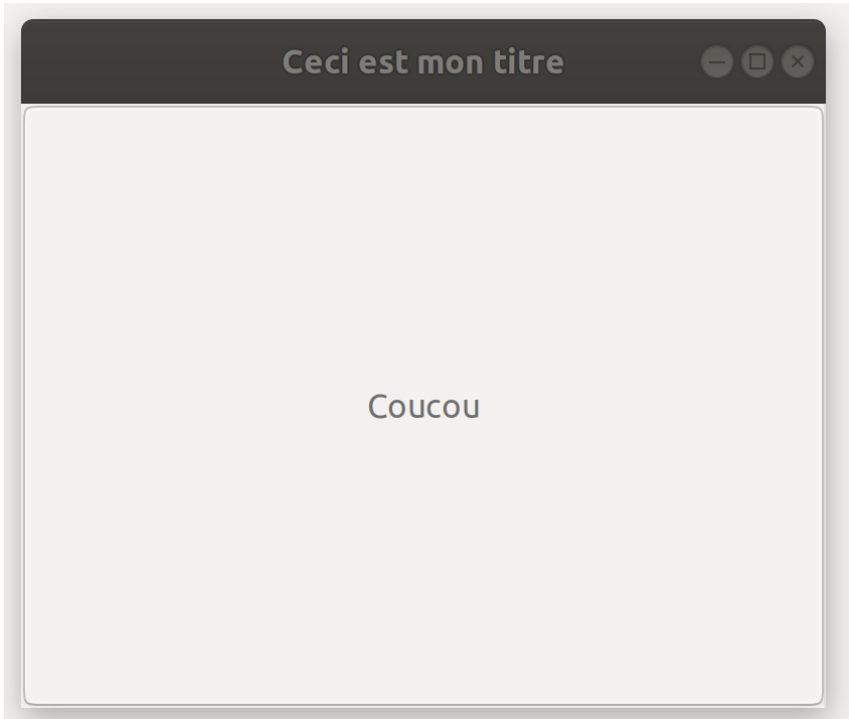
En résumé:

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

if __name__ == '__main__' :

    win = Gtk.Window()
    win.set_size_request(400, 300)
    win.set_title("Ceci est mon titre")

    win.connect("destroy", Gtk.main_quit)

    win.show_all()
    Gtk.main()
```

Boutons

On crée un bouton et on l'accroche à win:

```
b1 = Gtk.Button(label="Coucou")
win.add(b1)
```



Ce bouton prend toute la place et ne fait rien.

Boutons

On peut préciser une action :

```
def b1_on_clicked (widget) :
    print ("Bouton b1 cliqué")

b1.connect ("clicked", b1_on_clicked)
```

La fonction que l'on passe en second paramètre s'appelle une "callback". On peut lui passer une donnée quelconque (mécanisme de GTK+) :

```
def b1_on_clicked (widget, data) :
    print ("Bouton 1 cliqué, data =", data)

b1.connect ("clicked", b1_on_clicked, "Plop")
```

On peut passer un dictionnaire, un objet, etc.

Boutons

On peut utiliser un autre mécanisme de python pour passer des données : la callback reçoit le widget, or tout widget est de type class, donc on peut lui rajouter des attributs :

```
def b1_on_clicked (widget) :
        print ("Bouton 1 cliqué, titi =", widget.titi)

    b1.titi = "Plop"
```

Conteneurs

On ne peut accrocher qu'un seul widget dans un window. Pour en placer plusieurs il faut des conteneurs *Hbox* et *Vbox*.

```
vbox = Gtk.VBox()
win.add(vbox) # et pas win.add(bouton)
hbox = Gtk.HBox()
vbox.pack_start(hbox, expand=False, fill=False, padding=0)

b1 = Gtk.Button(label="Coucou")
hbox.pack_start(b1, expand=False, fill=False, padding=0)
b1.connect ("clicked", b1_on_clicked,"dit coucou")

b2 = Gtk.Button(label="Quit")
hbox.pack_start(b2, expand=Trure, fill=False, padding=0)
b2.connect ("clicked", Gtk.main_quit)
b3 = Gtk.Button(label="Gros bouton")
vbox.pack_start(b3, expand=True, fill=True, padding=0)
b3.connect ("clicked", b1_on_clicked,"dit gros bouton")
```

Zone de dessin DrawingArea

Création d'un DrawingArea :

```
darea = Gtk.DrawingArea()
```

On l'accroche dans la fenêtre, par exemple à la place du gros bouton, en lui donnant aussi toute la place :

```
vbox.pack_start(darea, expand=True, fill=True, padding=0)
```

Pour le moment le darea existe mais rien ne semble affiché. Il faut lui donner une callback pour dessiner :

```
def darea_on_draw (darea, cr) :
    print("Événement draw reçu")

darea.connect("draw", darea_on_draw)
```

Zone de dessin DrawingArea

On peut aussi lui confier une donnée avec un paramètre supplémentaire :

```
def darea_on_draw (darea, cr, data) :
    print("Événement draw reçu, data =", data)

darea.connect("draw", darea_on_draw, "Ploum")
```

De même en rajoutant des attributs à darea :

```
def darea_on_draw (darea, cr) :
    print("Événement draw reçu, toto =", darea.toto)

darea.connect("draw", darea_on_draw)
darea.toto = "Ploum"
```

Zone de dessin DrawingArea

Pour dessiner dans on_area_draw on utilise cr. Impossible de dessiner sans lui. Il est détruit et recréé à chaque appel. C'est la seule callback où on le reçoit, c'est donc le seul endroit où on peut dessiner !

```
def darea_on_draw (darea, cr) :
    print("Événement draw reçu")

    cr.set_source_rgb(0, 0, 1)           # Couleur RGB des prochains dessins
    cr.set_line_width(2)                 # Épaisseur des prochains dessins
    cr.set_font_size(14)                 # Taille des prochains textes

    cr.move_to(30, 100)                  # Commence un chemin
    cr.line_to(200, 120)                 # rajoute une ligne au chemin
    cr.line_to(260, 90)                  # rajoute une ligne au chemin
    cr.stroke()                          # Dessine le chemin et le supprime

    cr.move_to(20, 10+14)                # Commence un chemin
    cr.show_text("Ceci est du texte")    # Dessine du texte à cet endroit
    cr.stroke()                          # supprime le chemin

    cr.rectangle (100, 200, 50, 80)      # coin, largeur hauteur
    cr.stroke()

    cr.arc (180, 200, 40, 0, 2*math.pi) # centre, rayon, angles
    cr.stroke()
```

Zone de dessin DrawingArea

Principe:

1. le darea est automatiquement effacé à chaque appel
2. on paramètre cr (couleur, épaisseur, etc)
3. on crée des "chemins" invisibles
4. on les transforme en dessins avec cr.stroke(), ce qui supprime le chemin

Zone de dessin DrawingArea

On peut faire à la place un cr.fill() :

```
cr.arc (180, 200, 40, 0, 2*math.pi) # centre, rayon, angles
cr.fill()
```

Si on veut remplir dans une couleur et tracer le bord dans une autre couleur on peut préserver le chemin :

```
cr.arc (180, 200, 40, 0, 2*math.pi) # centre, rayon, angles
cr.set_source_rgb(0.8, 0.8, 0.8)
cr.fill_preserve()
cr.set_source_rgb(0, 0.8, 0)
cr.stroke()
```

Pour forcer darea à se redessiner depuis une callback, appeler

```
darea.queue_draw()
```

Pour cela il faut mémoriser darea au préalable dans le widget.