Méthodologie de la programmation

Pablo Rauzy
pr@up8.edu
pablo.rauzy.name/teaching/mdlp

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS
UFR MITSIC / L1 informatique

Séance 4

Les bibliothèques en Python
Pygame
Projets

- Apprendre à utiliser et écrire des bibliothèques logicielles en Python.
- Introduction de la bibliothèque PyGame.

Les bibliothèques en Python

- Le code Python est organisé en *modules*.
- Vous savez déjà écrire un module Python : il s'agit simplement d'un fichier avec du code Python dedans !
- Tout ce qui est défini dans le fichier fera parti du module.
- Pour utiliser un module, il suffit de faire appel à la directive **`import`** avec le nom du fichier (sans le `.py`).

Exemple

- Le fichier `foo.py` :

```
1 a = 42
2
3 def f (n):
4   return a + n
```

- Le fichier `bar.py` :

```
1 import foo
2
3 print(foo.a) # affiche "42"
4 print(foo.f(9)) # affiche "51"
```

- Le code qui est dans le fichier importé est exécuté, du coup on ne veut pas qu'il contienne autre chose que des définitions.
- La solution fourni par Python est une variable `__name__` qui vaut `"__main__"` quand le programme est appelé directement, ou le nom du module quand le programme est importé.
- Exemple avec le fichier `name.py` :

```
1 print("Le nom du module est : " + __name__)
```

  - Si on le lance comme un programme avec la commande `python3 name.py` :
    `Le nom du module est : __main__`.
  - Si on l'importe comme une bibliothèque avec **`import`** `name` :
    `Le nom du module est : name`.

Exemple

- Le fichier `fact.py` :

```
1  def fact (n):
2      result = 1
3      while n > 1:
4          result = n * result
5          n = n - 1
6      return result
7
8  if __name__ == '__main__':
9      import sys
10     print(fact(int(sys.argv[1])))
```

- On peut l'utiliser comme programme : `python3 fact.py 6`
- On peut l'utiliser comme bibliothèque : **`import`** `fact`

- Il est possible de rendre un script Python directement exécutable, sans avoir à le passer en argument de la commande `python3`.
- Pour cela il faut faire deux choses :
  1. marquer le fichier comme exécutable,
  2. indiquer avec quel interpréteur est chargé de son exécution.
- 1. se fait avec la commande `chmod` (comme “cHange MODe”).
  - `chmod +x` *`chemin-du-fichier`* rend exécutable (`+x`) le fichier donné.
- 2. se fait en mettant un *shebang* en première ligne du fichier.
  - Un shebang commence par `#!`, puis est suivi du chemin vers l’interpréteur : `#!/usr/bin/python3` par exemple.

Exemple

- Le nouveau fichier `fact.py` :

```
1  #!/usr/bin/python3
2
3  def fact (n):
4      result = 1
5      while n > 1:
6          result = n * result
7          n = n - 1
8      return result
9
10 if __name__ == '__main__':
11     import sys
12     print(fact(int(sys.argv[1])))
```

- On peut l'utiliser comme programme directement : `fact.py 6`
- On peut l'utiliser comme bibliothèque : **`import`** `fact`

- Si dans un autre script ou dans l'interpréteur on **`import fact`**, on peut maintenant utiliser la fonction `fact.fact` :

```
#!/usr/bin/python3

import fact
import random

def print_factorial_steps (n):
    for i in range(1, n+1):
        print('×'.join([str(x) for x in range(1, i+1)])
            + ' = ' + str(fact.fact(i)))

if __name__ == '__main__':
    n = int(input('Jusqu\'où afficher factorielle ? '))
    print_factorial_steps(n)
```

- Si dans un autre script ou dans l'interpréteur on **`import fact`**, on peut maintenant utiliser la fonction `fact.fact` :

```
#!/usr/bin/python3

import fact
import random

def print_factorial_steps (n):
    for i in range(1, n+1):
        print('×'.join([str(x) for x in range(1, i+1)])
            + ' = ' + str(fact.fact(i)))

if __name__ == '__main__':
    n = int(input('Jusqu\'où afficher factorielle ? '))
    print_factorial_steps(n)
```

- Il y a des cas comme ici où ça peut être embêtant d'écrire `fact.fact`.
- Il y a plusieurs solutions pour remédier à ce problème.

import … as …

- On peut renommer le module lors de l'import :

```python
#!/usr/bin/python3

import fact as f

def print_factorial_steps (n):
    for i in range(1, n+1):
        print('×'.join([str(x) for x in range(1, i+1)])
            + ' = ' + str(f.fact(i)))

if __name__ == '__main__':
    n = int(input('Jusqu\'où afficher factorielle ? '))
    print_factorial_steps(n)
```

- Cela peut permettre d'éviter des conflits de noms.

- On peut sélectivement importer directement certaines définitions :

```
1  #!/usr/bin/python3
2
3  from fact import fact
4
5  def print_factorial_steps (n):
6      for i in range(1, n+1):
7          print('×'.join([str(x) for x in range(1, i+1)])
8              + ' = ' + str(fact(i)))
9
10 if __name__ == '__main__':
11     n = int(input('Jusqu\'où afficher factorielle ? '))
12     print_factorial_steps(n)
```

- Il est possible d'importer plusieurs définitions en les séparant par des virgules, ou directement toutes les définitions avec un `*`.

- La *bibliothèque standard* de Python contient **beaucoup** de modules.
- On a vu lors du dernier TP le module `random` qui sert à la génération de données aléatoires.
- On a aussi vu dans ce cours le module `sys`, qui a notamment le tableau `argv` avec les arguments de la ligne de commande.
- Il y a aussi des fonctions mathématiques dans `math`.
- La documentation de la bibliothèque standard est disponible en ligne : `https://docs.python.org/3/library/`.
- La version de Python installée sur les machines du Bocal est la 3.4.2.

Pygame

- On va étudier une grosse librairie en particulier : Pygame.
- Pygame sert principalement à faire des jeux vidéos.
- Site web : `https://pygame.org/`.
- Documentation : `https://pygame.org/docs/`.

- *Logiciel libre*.
- Multi-plateformes : GNU/Linux, Windows, Mac OS X.
- Respecte bien les conventions Python.

- Lorsqu'on programme un jeu il faut gérer :
  - l'affichage,
  - l'interaction avec l'utilisateur,
  - le temps,
  - (parfois) le réseau.

- Dans un jeu, l'affichage est critique.
- Beaucoup de composants à gérer tout en restant fluide.
- Utilisation de la technique du *double buffer*.
- Automatique dans les bibliothèques haut niveau comme Pygame.

Le double buffer

- Si on essaye d'afficher plein d'éléments à l'écran, un scintillement peut se produire et casser la fluidité de l'affichage.
- La technique du double buffer consiste à avoir deux zones d'affichage, et d'alterner entre les deux.
- De cette manière, on dessine toujours sur la zone "cachée", et aucun artefact gênant ne peut arriver à l'écran.
- Chaque fois qu'on a fini de dessiner, on inverse les deux zones et celle qu'on vient de dessiner est affichée d'un seul coup.

Le double buffer avec Pygame

- La bibliothèque Pygame fait tout ça pour vous si vous le lui demandez :

```
1  import sys
2  import pygame
3  from math import cos
4
5  pygame.init()
6
7  size = width, height = 800, 100
8  screen = pygame.display.set_mode(size, pygame.DOUBLEBUF)
9  black, white = pygame.Color(0,0,0), pygame.Color(255,255,255)
10 screen.fill(white)
11
12 i = 0
13 while True:
14     for event in pygame.event.get():
15         if event.type == pygame.QUIT:
16             sys.exit()
17
18     y = int((height / 2) * cos(i / 50) + (height / 2))
19     x = i % width
20     i += 1
21     screen.set_at((x, y), black)
22     pygame.display.flip()
```

- Il est évidemment mieux de faire le moins d'opérations d'affichage possible.
- Par exemple dans un jeu de plateau, il est possible de construire le plateau au début du niveau puis de le garder en mémoire déjà construit pour l'afficher d'un seul coup.
- Pour les autres composants du jeu, par exemple une balle en mouvement qui va systématiquement bouger, on peut les mettre dans un conteneur qui va permettre d'itérer dessus pour les afficher.

```
1 bg = pygame.Surface(size, pygame.SRCALPHA, 32)
2 pygame.draw.line(bg, black, (30,20), (30,100), 10)
3 pygame.draw.line(bg, black, (30,100), (70,100), 10)
4 pygame.draw.line(bg, black, (80,50), (120,20), 10)
5 pygame.draw.line(bg, black, (120,20), (120,100), 10)
6     # update display
7     screen.blit(bg, (0, 0))
```

- De la même manière, il faut mieux éviter de charger plein de petites images.
- On préfère donc charger une seule grosse image qui sert pour plusieurs *sprites*.

```
player = pygame.image.load('player.png').convert()
player.set_colorkey(white)
up = pygame.Rect(25, 0, 25, 31)
down = pygame.Rect(0, 0, 25, 31)
left = pygame.Rect(25, 31, 25, 31)
right = pygame.Rect(0, 31, 25, 31)
    # update display
    screen.blit(player, position, direction)
```

- Les entrées utilisateurs arrivent par la souris, le clavier, et/ou un joystick, dans le cas des jeux.
- La gestion de ces périphériques repose sur le principe d'évènements.
- Chaque fois que l'utilisateur est actif, un évènement est généré :
  - mouvement de la souris,
  - clic du bouton gauche / droit / milieu,
  - appui sur une touche,
  - ...
- Quand c'est applicable, l'évènement contient des informations utiles :
  - position du curseur de la souris,
  - quelle touche du clavier a été enfoncée,
  - ...

Boucle de gestion des évènements

- La gestion des évènements doit se faire en continue.
- Elle est donc placée au début de la boucle principale du programme.
- La structure de la boucle principale d'un jeu va être :

```
done = False
while not done:
    for event in pygame.event.get():
        # manage events
    # clear screen
    screen.fill(white)
    # update display
    screen.blit(bg, (0, 0))
    screen.blit(player, position, direction)
    # flip double buffer
    pygame.display.flip()
```

Gestion des évènements

- Pygame fourni une gestion complète et simple des évènements.
- La méthode `pygame.event.get()` renvoie la liste des évènements en attente.
- Chaque évènement a un `type` et contient des informations additionnelles.
  - Un évènement souris contiendra aussi les boutons cliqués et la position du curseurs.
  - Un évènement clavier contiendra aussi la touche appuyée et d'éventuels modificateurs.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        done = True
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_UP:
            direction = up
            position[1] -= 2
        elif event.key == pygame.K_DOWN:
            direction = down
            position[1] += 2
        elif event.key == pygame.K_LEFT:
            direction = left
            position[0] -= 2
        elif event.key == pygame.K_RIGHT:
            direction = right
            position[0] += 2
```

- Regardons ensemble le code que l'on vient d'écrire et ce qu'il produit...

Projets

- Seul, ou à deux exceptionnellement (gros projet).
- Programmation d'un jeu en 2D.
- Rendre un premier rapport par email pour mi-novembre.
- Rendre le code du jeu et le rapport final pour le mi-décembre.
- Les dates exactes sont données sur la page web du cours.

- En Python, avec Pygame.
- Le code doit être propre !
- Vous devez être capable de m'expliquer son fonctionnement dans les moindre détails.

- Les rapports doivent être en PDF.
- Le rapport intermédiaire :
  - Décrire le jeu.
  - Expliquer les choix prévus d'architecture logicielle.
  - Si à deux, donner la répartition prévue des tâches.
- Le rapport final :
  - Expliquer l'organisation du code.
  - Expliquer les difficultés rencontrées et comment vous les avez surmontées ou contournées.
  - Si à deux, qui à fait quoi.

Bonus

- Historique de développement propre fourni avec Git.
- Rapports produits avec LaTeX.
- Des séances de ce cours auront pour sujet Git et LaTeX dans la seconde moitié d'octobre.

- Quelques familles de jeu :
  - Tetris
  - Casse briques
  - Bubble shooter
  - Pacman
  - Jewel
  - Copter
  - Tanks
  - 2048
  - Snake
  - Flipper
  - Tron
  - Sokoban
  - Plateforme (Mario-like)
  - RPG (Pokemon-Like)
  - ...