ICC: Programmation

GC/MX, Cours 11, 29 novembre 2019

Jean-Philippe Pellet

Previously, on Programmation…

- Types de base en Python: int, float, str, bool
- Méthodes, fonctions et slicing pour calculer des valeurs dérivées
- Conditions pour exécuter du code selon la valeur d'une expression booléenne
- Boucles pour exécuter du code plusieurs fois:
- Déclaration de fonctions avec type de retour et paramètres
- Utilisation de listes, sets, dictionnaires, tuples
- Déclaration de classes simples et de méthodes dans ces classes
- Création d'interfaces graphiques simples avec Tkinter
  - Fenêtres, Frames, geometry manager, widgets
- Fonctions comme valeurs, paramètres…
  - Fonctions d'ordre supérieur
  - Fonctions lambda

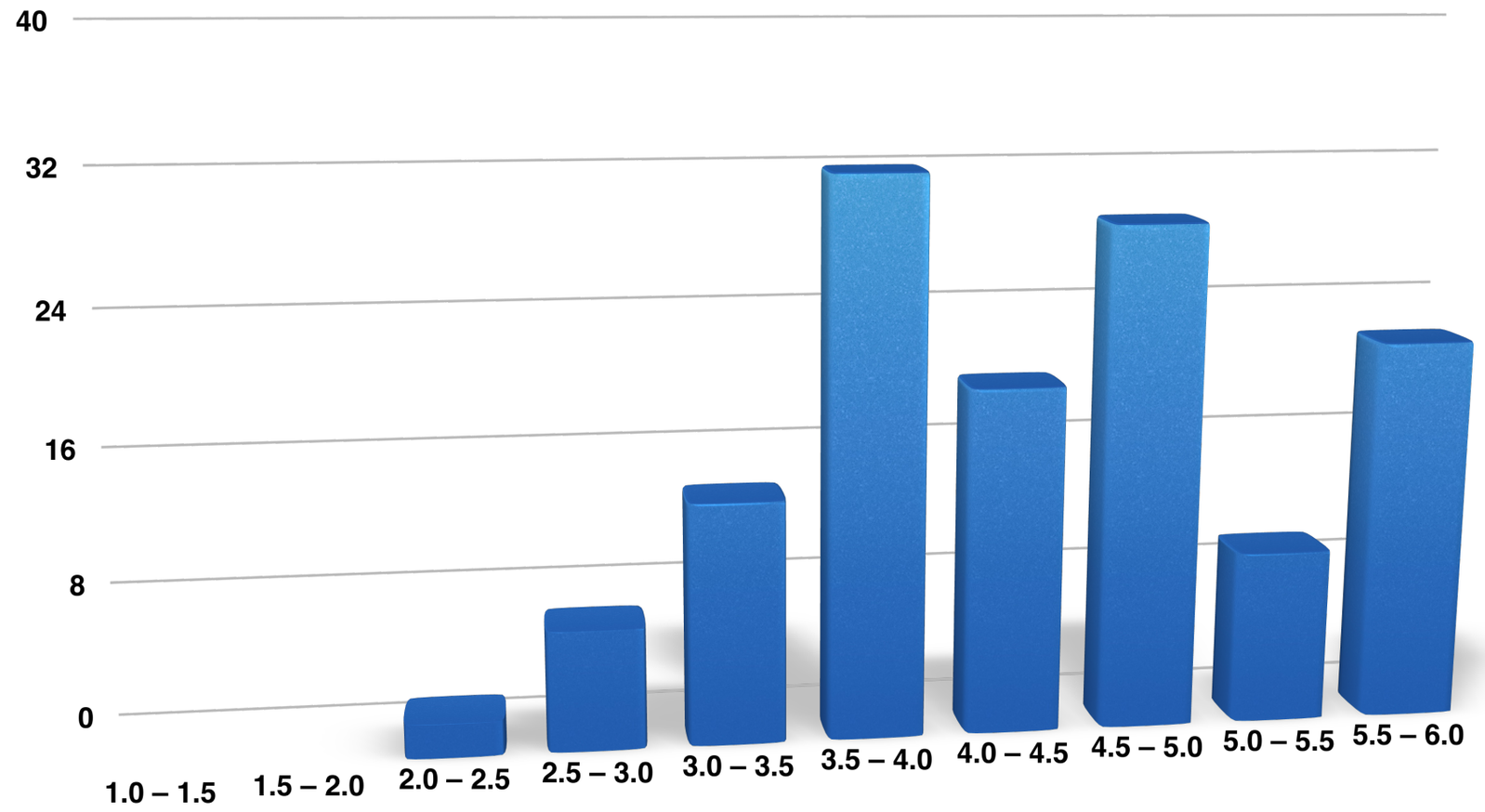
Examen intermédiaire

- Échelle: **Note indicative** *n* en fonction du nombre de points *p*

  $n = \min(6, \text{round}((p / (115 / 5.0) + 1), 2))$

  ➡ Note minimale **1**, maximale **6,** arrondi au centième
  *(note finale arrondie **au quart**)*

- Moyenne: **4.37**
- Médiane: **4.30**
- Mode: **3.5–4.0**

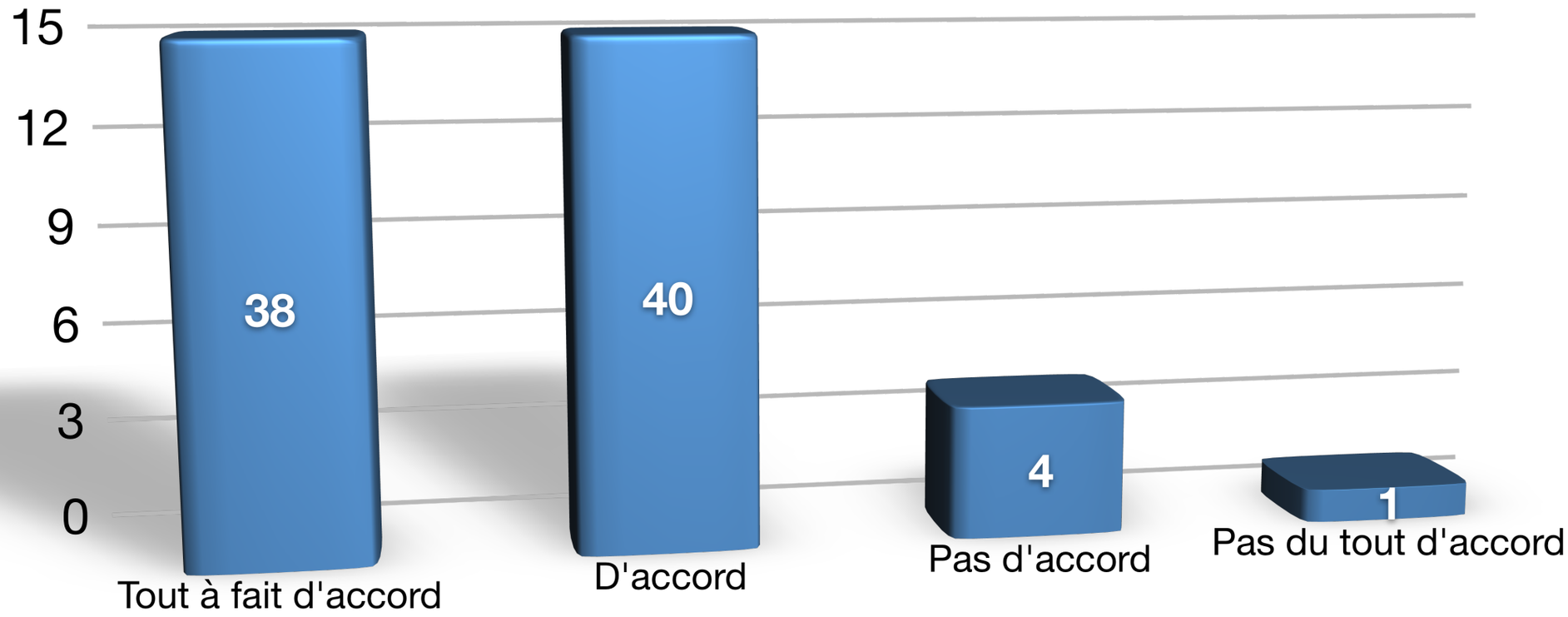


Examen intermédiaire

- Corrigé en ligne
  - Lisez-le et comprenez bien chaque ligne
  - Pas la *seule* solution correcte!
- Votre décompte détaillé de points: dans la rubrique *Grades* sur Moodle, y compris *rang centile*, entre 0 et 100
- Discussion du corrigé: sur le forum
- Discussion de la correction: avec moi (mais pas de «pêche aux points» svp.)
- Note finale:
  - **30%** examen intermédiaire théorique
  - **30%** cet examen intermédiaire,
  - **40%** examen final commun sur papier

Évaluation du cours

«Dans l'ensemble, je pense que ce cours est bon.»



Pourcentages cumulés: **45%** **94%** **99%**

Participation: 83/141, dont 41 avec remarque

Merci pour votre feedback!

C'est un plaisir pour moi de donner ce cours dans de telles conditions!

Feedback *(partiel; tel quel; biaisé pour montrer les critiques! :))*

Pour la partie programmation, lors des exercices **des salles plus grandes** et donc plus adaptées au nombre d élèves seraient souhaitables.

Les cours sont d'un **très bon niveau**. Les deux professeurs sont sans conteste **passionnés** et cela se ressent dans la qualité des cours. Je n'avais jamais fait d'informatique et algorithmique auparavant, et malgré la difficulté que j'ai pu rencontrer au début du semestre, les cours m'ont **réellement permis d'apprendre** comment m'en sortir! Je me permets seulement de préciser que pour la partie programmation, le projet "Cargobot" est **délicat** et les exercices pour le mettre en oeuvre sont assez compliqués même avec l'aide des tuteurs et du professeur...

Le cours est **très bon**, mais le problème principale est que cette matière nécessite beaucoup **trop de temps** pour le temps de travail qu'elle demande alors qu'elle est la moitié d'un cours du petit bloc

---

Même si la matière **ne m'intéresse pas du tout**, nos deux professeurs **font tout pour nous intéresser** et sont vraiment passionnés par leur branche.

---

programmation : Certaines notions dans les exercices ne sont pas abordées en cours, donc c'est parfois un peu **difficile de s'y retrouver**

---

programmation: le cours est très clair et les **démos** aident à bien comprendre les notions

---

La partie pratique **ne donne pas les outils nécessaires** pour réussir le midterm en n’ayant jamais utilisé python avant.

Programmation: les **démo** sur python sont supers.

Professeurs très compétents qui **expliquent bien**.

Les cours sont **clairs et bien expliqués**. Les corrigés sont des fois un peu **difficiles à comprendre**. Aussi , j’ai personnellement trouvé qu’on avançait très vite et que lors du midterm, on exigé un niveau assez difficile à avoir pour ceux qui n’ont jamais programmer avant de venir a l’epfl.

Prog: **commenter la correction** avec des # pr expliquer ce que vous faites. Meilleure organisation des salles d’exercice. Sinon très bien, vous commencez tt les 2 par les **bases**.

Cours de cette semaine

Classes et sous-classes
Introduction à l'héritage simple

Exemple suivi

Tâche: modélisation d'un tableau interactif (simple). On doit pouvoir représenter des cercles, rectangles, et textes à des positions données. On commence simple:

```
class Point:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    def __repr__(self) -> str:
        return f"({self.x}, {self.y})"

    def translateBy(self, dx: float = 0, dy: float = 0):
        self.x += dx
        self.y += dy
```

Ces arguments ont des valeurs par défaut

Création d'une instance de *Point*

Modification d'un champ de *p*

Appel de méthode (avec arguments nommés; l'argument *dx* a la valeur par défaut de 0)

```
p = Point(1, 2.5)
print(p)  # (1, 2.5)
p.x += 10
print(p)  # (11, 2.5)
p.translateBy(dy=2.5)
print(p)  # (11, 5)
```

Classes et sous-classes

On déclare une classe pour modéliser un rectangle positionné dans le plan.

```
class Rectangle:
    def __init__(self, center: Point, width: float, height: float):
        self.center = center
        self.width = width
        self.height = height

    def __repr__(self) -> str:
        return (
            f"Rectangle(center={self.center}, width={self.width}, “
            f"height={self.height})"
        )

    def translateBy(self, dx: float = 0, dy: float = 0):
        self.center.translateBy(dx, dy)
        self.center.y += dy
```

Good to know: la juxtaposition de deux strings entre guillemets les concatène

On peut directement modifier les champs *x* et *y* du *center*, ou…

… on appelle la méthode *translateBy*, aussi définie pour un Point!

```
r = Rectangle(p, 3, 4)
print(r) # Rectangle(center=(11, 5.0), width=3, height=4)
```

Modélisation

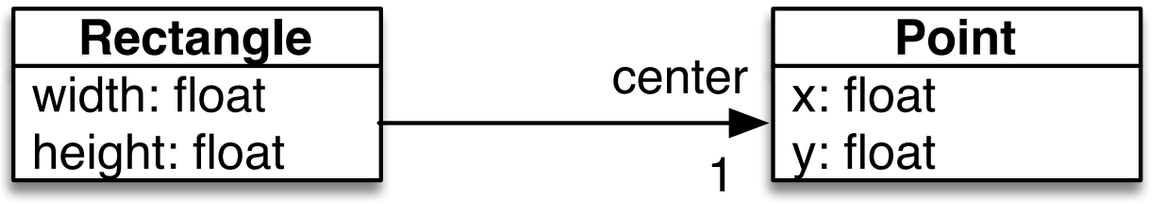
Conceptuellement, ici, un rectangle est défini par un point et par une largeur et une hauteur. Comme le point est lui-même défini par deux données x et y, le rectangle est donc défini par 4 floats séparés.

Point
x: float
y: float

Rectangle
center: Point
width: float
height: float

Rectangle
centerX: float
centerY: float
width: float
height: float

Notation: Diagramme de classe UML



Classe *Circle*

```python
class Circle:
    def __init__(self, center: Point, radius: float):
        self.center = center
        self.radius = radius

    def __repr__(self) -> str:
        return (
            f"Circle(center={self.center}, "
            f"radius={self.radius})"
        )

    def translateBy(self, dx: float = 0, dy: float = 0):
        self.center.translateBy(dx, dy)


c = Circle(Point(20, 20), 10)
print(c) # Circle(center=(20, 20), radius=10)
```

Classe *Text*

```
class Text:
    def __init__(self, center: Point, text: str):
        self.center = center
        self.text = text

    def __repr__(self) -> str:
        return (
            f"Text(center={self.center}, "
            f"text={repr(self.text)})"
        )

    def translateBy(self, dx: float = 0, dy: float = 0):
        self.center.translateBy(dx, dy)


t = Text(Point(1, 1), "hello there!")
print(t) # Text(center=(1, 1), text='hello there!')
```

Élimination du code répété

```
class Rectangle:
    def __init__(self, center: Point, width: float, height: float):
        self.center = center
        self.width = width
        self.height = height

    def __repr__(self) -> str:
        return (
            f"Rectangle(center={self.center}, width={self.width}, “
            f"height={self.height})"
        )
```

On a écrit presque le même code dans ces trois classes!

```
def translateBy(self, dx: float = 0, dy: float = 0):
    self.center.translateBy(dx, dy)
```

On peut isoler le (ou une partie du) code commun dans une superclasse

Superclasse, sous-classe

Conceptuellement: comme si les champs et les méthodes de la superclasse étaient copiés dans la sous-classe

```
class BoardElement:
    def __init__(self, center: Point):
        self.center = center

    def translateBy(self, dx: float = 0, dy: float = 0):
        self.center.translateBy(dx, dy)
```

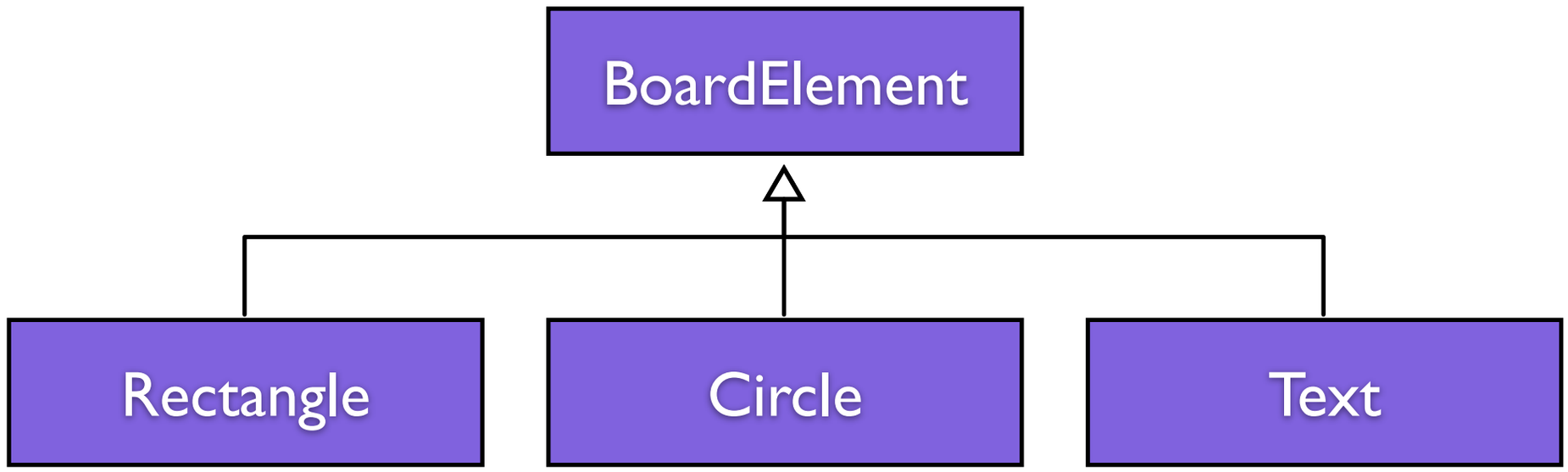
Déclaration de la superclasse

```
class Rectangle(BoardElement):
    def __init__(self, center: Point, width: float, height: float):
        BoardElement.__init__(self, center)
        self.width = width
        self.height = height

    def __repr__(self) -> str:
        return (
            f"Rectangle(center={self.center}, width={self.width}, "
            f"height={self.height})"
        )
```

Appel du __init__ de la superclasse

Héritage



- *Rectangle*, *Circle* et *Text* héritent de *BoardElement*
- Toutes les méthodes définie par *BoardElement* sont disponibles sur les objets de type *Rectangle*, *Circle* ou *Text*
- Les champs sont aussi hérités
- La méthode __init__ des sous-classes doit appeler le __init__ de la superclasse

Syntaxe

- Déclaration de la superclasse:

```
class A:
    ...

class B(A):
    ...
```

Entre parenthèses, nom de la superclasse (préalablement déclarée). Il peut y avoir plusieurs superclasses

- Appel du __init__ de la superclasse:

```
class B(A):
    def __init__(self, ...):
        A.__init__(self, ...)
        self...
```

Méthode __init__ normale…

… mais qui appelle la méthode __init__ de la superclasse A en répétant le paramètre *self*

Elle peut ensuite continuer à faire son travail, par exemple assigner de nouveaux champs, comme d'habitude

Liste de *BoardElements*

Démo

Liste de *BoardElements*

```
elements: List[BoardElement] = [
    Text(center=Point(10, 10), text="hello"),
    Circle(center=Point(0, 10), radius=5),
    Rectangle(center=Point(50, 20), width=30, height=40),
]

print("Éléments sur le tableau:")
for elem in elements:
    print(elem)

for elem in elements:
    elem.translateBy(dx=10, dy=-10)

print("--")
print("Après translation:")
for elem in elements:
    print(elem)
```

Création normale d'instances des sous-classes

BoardElement est le type commun le plus spécifique (*LUB* pour *least upper bound*)

Rappel: affichage de l'objet, print va utiliser *__repr__*

Bien que chaque objet ait un type différent, on peut appeler cette méthode définie dans la superclasse *BoardElement* qui fera la même chose

Pourquoi l'héritage?

- Structuration du code. Si *A* hérite de *B*, ça doit être vrai que «*A* est une sorte de *B*» (*Circle* est une sorte spéciale de *BoardElement*)
- Réutilisation du même code. On évite de retaper le même code dans plusieurs sous-classes
- Présentation d'une interface unique aux utilisateurs de votre code sans qu'ils aient besoin de connaître le détails des sous-classes

  — *L'interface est la série de méthodes (ou de champs) que les utilisateurs de vos classes «voient»*

Factorisation et réutilisation du code

- **Boucles**: éviter de taper x fois le même code *(surtout quand x n'est pas connu lors de la compilation)*
- **Listes**: éviter de déclarer x variables du même type *(même remarque pour x)*
- **Fonctions et méthodes**: donner un nom à une série d'instructions et éviter de dupliquer du code à plusieurs endroits différents
- **Superclasses**: éviter de déclarer plusieurs fois les mêmes structures (méthodes ou champs) dans des classes différentes

Résumé Cours 11

- Selon vous, le cours n'est globalement pas trop mauvais :)
- Une classe peut avoir une superclasse déclarée entre parenthèses après le nom de la classe
- Une sous-classe hérite des méthodes et des champs de la superclasse et doit traditionnellement appeler la méthode __init__ de sa superclasse