

# Programmation

SIE/GCG, Cours 10

29 avril 2024

Jean-Philippe Pellet

```

class ProgramView(Canvas):
    def __init__(self, parent, size) -> None:
        Canvas.__init__(self, parent, height+2 * TOP_MARGIN + 4 * LINE_HEIGHT, highlightthickness=0, size)

    def redraw(
        self,
        program: Program,
        current_subprogram: List[Instruction],
        current_instruction_index: int,
    ) -> None:
        height = self.winfo_height()
        width = self.winfo_width()

        self.delete(ALL)
        self.create_rectangle(0, 0, width, height, fillwindow_background_color, width=0)

        # boucle pour les 4 sous-programmes P1 à P4
        for i, subprogram in enumerate(
            [program.P1, program.P2, program.P3, program.P4]
        ):
            # dessin du titre
            instruction_center_y = TOP_MARGIN + 1 * LINE_HEIGHT + LINE_HEIGHT // 2
            self.create_text(LEFT_MARGIN // 3, instruction_center_y, text=f"P{i + 1}")

            # dessin de chaque instruction
            for j, instr in enumerate(subprogram):
                instruction_center_x = (
                    LEFT_MARGIN
                    + j * (INSTRUCTION_BOX_SPACING + INSTRUCTION_BOX_WIDTH)
                    + INSTRUCTION_BOX_WIDTH // 2
                )
                instruction_center_x = instruction_center_x - INSTRUCTION_BOX_WIDTH // 3

```

# Previously, on ICC/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:  
`if <condition>: ... else: ...` et ses variantes
- **Boucles** pour exécuter du code plusieurs fois:
  - Boucle `while <condition>: ...`
  - Boucle `for i in range(...): ...`
- **Déclaration de fonctions** avec type de retour et paramètres:
  - `def calculate_area(r: float) -> float: return ...`
- Utilisation de **listes, sets** et **dictionnaires**
- Déclaration de **classes**: `@dataclass class Rectangle: ...`
- Création, chargement, manipulation et sauvegarde d'**images**
- **Programmation dynamique** pour trouver des seams

# Miniprojet

---

- **Inscription aux groupes** en ligne
  - Délai d'inscription: avant lundi prochain 6 mai
- Résultats attendus comme images: sur Moodle
- **Délai de rendu: 31 mai 2024 à 23h59**
  - Je vous encourage à le faire avant!
- Un seul fichier à rendre: `miniproject.py`
- **Un seul rendu** par groupe (membre I du groupe)
- Evaluation: uniquement de vos **fonctions à compléter**
  - Pas du “main”

# Fin du semestre

---

- Aujourd'hui
  - Exercices: courts pour temps miniprojet
  - Corrections vidéos!
- **6 mai**: lecture et écritures de fichiers texte
- **13 mai**: threads et données partagées
  - Je serai absent; vidéo de l'année précédente
- **20 mai**: Pentecôte, *pas de cours*
- **27 mai**: *dernier cours*, révisions selon vos souhaits
  - Sondage en ligne pour collecter thématiques
- **24 juin**: examen final

# Cours de cette semaine

*Compréhensions de listes*  
*Fonctions d'ordre supérieur*  
*Lambdas*

# Cours de cette semaine

*Compréhensions de listes*  
*Fonctions d'ordre supérieur*  
*Lambdas*

# Liste dérivée et compréhensions de listes

Créez une liste de int qui indique la taille de chaque string issu d'une liste de strings donnée

```
words: list[str] = ["Elvis", "has", "left", "the", "building"]
```

```
size_of_words: list[int] = []
```

```
for word in words:
```

```
    size = len(word)
```

```
    size_of_words.append(size)
```

— On prépare une liste initialement vide

Pour chaque mot de la liste, on obtient sa longueur et on ajoute cette valeur à notre nouvelle liste

```
print(words)
```

```
# ['Elvis', 'has', 'left', 'the', 'building']
```

```
print(size_of_words)
```

```
# [5, 3, 4, 3, 8]
```

Compréhension de liste:

```
size_of_words = [len(word) for word in words]
```

«une liste formée par le résultat de l'expression `len(word)` pour chaque mot de la liste `words`»

# Compréhensions de listes

[expression **for** x **in** container **if** condition]

expression (qui peut utiliser *x*) qui sera évaluée pour créer chaque élément de la liste

variable de boucle qui prendra successivement toutes les valeurs des éléments contenus

range(), liste, set, etc.

filtre avec if (optionnel)

- Moyen concis de **définition de listes**
  - L'expression de gauche se lit plutôt **en dernier**
- Plusieurs **for** enchaînables
- Possible de **filtrer** des itérations avec des **if**
- *Fonctionne aussi avec des **sets** et des **dictionnaires***

# Compréhensions de listes: exemples

```
[0 for _ in range(10)]  
# [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

variable non utilisée: 10 × la même valeur, 0

```
my_ints = [2, 2, 5, 6, 1, 6, 3, 2]  
[x for x in my_ints if x > 3]  
# donne [5, 6, 6]
```

filtrage à la volée avec un *if*

```
firsts = ["Jean", "Pierre"]  
seconds = ["Pierre", "Michel", "Marc", "Jean"]  
[f"{f}-{s}" for f in firsts for s in seconds if f != s]  
# ['Jean-Pierre', 'Jean-Michel', 'Jean-Marc',  
  'Pierre-Michel', 'Pierre-Marc', 'Pierre-Jean']
```

deux *for-in* successifs: combinaisons des variantes, ici en plus avec un filtre

```
[[x * y for y in range(1, 11)] for x in range(2, 11)]  
# [[2, 4, 6, ..., 18, 20], [3, 6, ..., 30], ..., [10, 20, ..., 100]]
```

génération de listes de listes

# Cours de cette semaine

*Compréhensions de listes*  
*Fonctions d'ordre supérieur*  
*Lambdas*

# Fonctions: des valeurs comme les autres

---

- Nous avons utilisé des variables et paramètres pour **stocker des valeurs «simples»**
  - int, float, bool
  - List, Dict, Set; np.array pour les images
  - Des instances de nos propres classes (OilSpillEvent, PixelData)
- On peut aussi les utiliser pour stocker et manipuler directement des **fonctions**

*Démo*

# Des variables contenant des fonctions

```
def square(x: int) -> int:  
    return x * x
```

Fonction «normale»; appel «normal»

```
print(square(4))
```

Variable affectée avec comme valeur... la fonction elle-même — sans appel avec «(arg)» à la fin

```
au_carré: Callable[[int], int] = square  
print(au_carré(4))
```

Type de quelque chose qui «peut être appelé», comme une fonction

```
def square2(x: int) -> int:  
    return int(math.pow(x, 2))
```

La fonction dans la variable se comporte et s'appelle comme une fonction «normale»

```
def square3(x: int) -> int:  
    return x ** 2
```

On peut par exemple mettre 3 fonctions dans une liste pour les manipuler dans une boucle

```
all_functions: list[Callable[[int], int]] = [square, square2, square3]  
for f in all_functions:  
    print(f(4))
```

Type: une liste de fonctions qui acceptent un int et retournent un int

# Le type Callable[[...], ...]

- Ce qui **peut être appelé** avec «(arguments)»
- **Plusieurs types** pour les arguments, un **type de retour**
- Exemples:
  - `f: Callable[[], int] = ...`
    - ➔ `f` s'appelle sans paramètre et retourne un `int`:  
`n: int = f()`
  - `f: Callable[[int], int] = ...`
    - ➔ `f` s'appelle avec un `int` comme paramètre et retourne un `int`:  
`n: int = f(42)`
  - `f: Callable[[str, int], set[str]] = ...`
    - ➔ `f` s'appelle avec un `string` comme premier paramètre, un `int` comme second, et retourne un ensemble de `strings`:  
`strings: set[str] = f("test", 5)`

# Des fonctions comme paramètres

```
def apply_twice(f: Callable[[int], int], value: int) -> int:  
    return f(f(value))
```

```
print(apply_twice(square, 4))
```

⇒ `print(square(square(4)))`

- Cette fonction accepte comme premier paramètre **une fonction**
- Elle l'utilise pour l'**appliquer deux** fois de suite à son argument
- Le type est en fait **trop restrictif...**
  - On pourrait très bien passer chaque fois `str` à la place, par exemple!
  - *Il y a des moyens en Python de faire ceci, que nous ne discutons pas*

# Des fonctions qui retournent des fonctions

- Des fonctions peuvent **accepter** des fonctions
- Elles peuvent aussi **retourner** des fonctions!

## Démo

```
def twice(f: Callable[[int], int]) -> Callable[[int], int]:  
    def f_of_f(x: int) -> int:  
        return f(f(x))  
    return f_of_f
```

```
fourth_power = twice(square)  
print(fourth_power(4))
```

La fonction *twice* accepte une fonction *f* et retourne une fonction qui est le résultat de la double application de *f*

Ici, *fourth\_power* est donc une fonction qui accepte un *int* et qui retourne un *int*, et qui s'appelle comme n'importe quelle autre fonction de premier ordre qu'on connaît déjà

# Cours de cette semaine

*Compréhensions de listes*  
*Fonctions d'ordre supérieur*  
*Lambdas*

# Fonctions lambda (1/2)

---

```
def plus_2(x: int) -> int:  
    return x + 2
```

```
plus_4 = twice(plus_2)
```

- Ici, on **nomme** la fonction `plus_2` pour la passer à `twice` comme argument
- Peut-on faire plus concis?
- $\Rightarrow$  **Fonction «anonyme», lambda**

```
plus_4 = twice(lambda x: x + 2)
```

# Fonctions lambda (2/2)

- Les fonctions lambda... `lambda x: x + 2`
  - N'ont pas de nom
  - Déclarent une liste d'arguments (sans type) avant le deux-points
  - Ont à droite du deux-points une seule expression
  - Ont un return automatique

```
lambda x, y: x * y
```

Fonction qui accepte deux paramètres et retourne leur produit

```
lambda: 42
```

Fonction qui n'accepte aucun paramètre et retourne 42

```
lambda x:  
    y = x + 2  
    return y * y
```

Pas possible sur plusieurs lignes; utilisez une fonction normale.

# Exemple: comme critère de tri

```
text = ... # un texte à analyser

def freq_analysis(text: str) -> dict[str, int]: ...
    ... # un dictionnaire qui relie chaque lettre à son nombre d'occurrences

frequencies: dict[str, int] = freq_analysis(text)
frequencies_as_tuples: list[tuple[str, int]] = list(frequencies.items())

sorted_frequencies: list[tuple[str, int]] = sorted(
    frequencies_as_tuples, key=lambda t: -t[1]
)

print(sorted_frequencies)
```

Création du dictionnaire  
(cf. un cours précédent)

Conversion du dictionnaire en une  
liste de paires (non triées)

Tri de la liste de paires, en utilisant comme clé de tri le  
*second* élément de la paire (le nombre d'occurrences de  
la lettre). Sans cela, tri selon l'ordre alphabétique... *key*  
est donc une fonction appliquée à chaque élément à trier

# Critère de tri

---

```
sorted(frequencies_as_tuples, key=lambda t: -t[1])
```

- Le paramètre `key` est une fonction qui retourne, pour chaque élément de la liste à trier, la **valeur selon laquelle trier** cet élément
- Nous voulons trier selon le **nombre d'occurrences** de la lettre
- Ici, on retourne **la case 1 du tuple**, donc le nombre d'occurrences
- On l'**inverse** pour trier dans l'ordre décroissant

# Résumé Cours 10

---

- On peut facilement déclarer des listes dérivées avec la syntaxe des **compréhensions de listes**
  - Un ou plusieurs *for*; 0, 1 ou plusieurs filtres avec *if*
- Les fonctions peuvent aussi être stockées dans des **variables**
- Elles peuvent être **manipulées** comme les autres valeurs
  - Passées en **paramètres**
  - **Retournées** depuis d'autres fonctions
- Les **fonctions lambdas** sont de courtes fonctions anonymes
  - Pratiques à déclarer de manière **succincte**, spécialement comme paramètre d'une autre fonction
  - Pratique comme argument pour **sorted**, par exemple