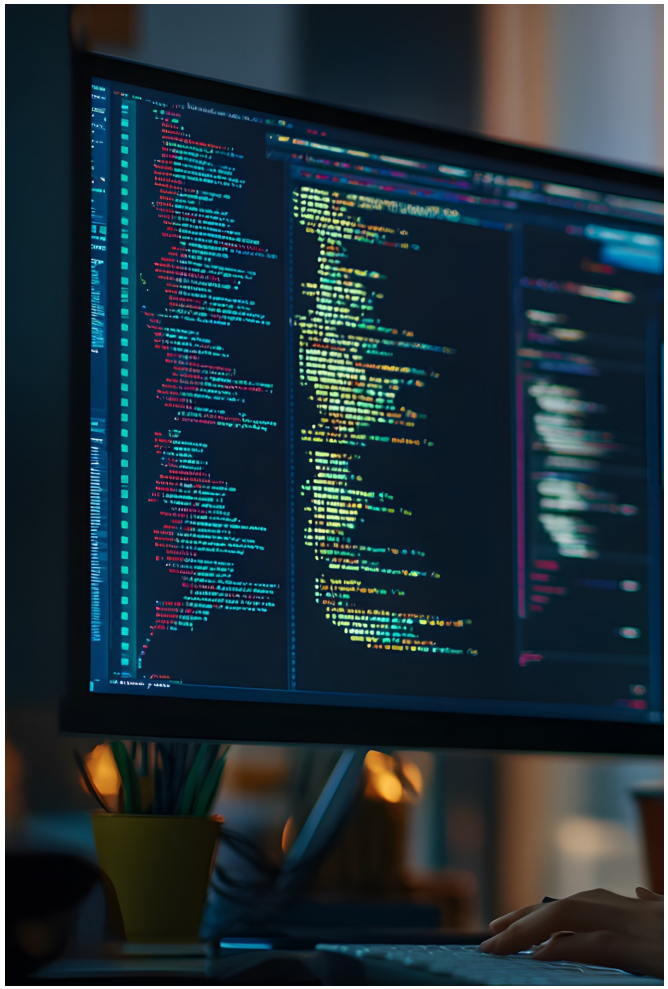
Information, Calcul et Communication

CS-119(k) ICC – Programmation
Semaine 4

Rafael Pires
rafael.pires@epfl.ch

Lausanne, 10.03.2025

Précédemment, dans... ICC-P



- Types de base en Python : `int`, `float`, `str`, `bool`
- Méthodes, fonctions et slicing
  pour calculer des valeurs dérivées
- Branchements pour exécuter du code selon la valeur d'une expression booléenne
- Boucles pour exécuter du code plusieurs fois
  - Interruptions et saut dans les boucles avec `break` et `continue`
- Fonctions

```
def nom_fonction(param1: type1, param2: type2, ...) -> type_de_retour:
    <instructions>
    return valeur_de_retour
```

Valeurs par défaut des paramètres

- Si les paramètres **to** et **n** ne sont pas précisés lors de l'appel, on utilise les valeurs standards

```
def say_hello(to: str = "John", n: int = 1) -> None:
    hello = " hello" * n
    print(f"Oh,{hello}, {to}!")

say_hello()                     # valeurs pas précisée: on utilise to="John" et n=1
say_hello("Peter")              # le premier paramètre est précisé, et n=1
say_hello(to = "Jane")          # même effet ici
# say_hello(3)                  # impossible, le premier paramètre est de type str
say_hello(n = 2)                # OK, car le paramètre est nommé
say_hello(n = 3, to = "James") # on peut réordonner les paramètres nommés
```

Conseil : Utilisez des noms clairs et des types

```python
def f(a, b):
    return a * b / 2

x = 5
y = 10
z = f(x, y)
print(z)
```

```python
def aire_triangle(base: float, hauteur: float) -> float:
    return base * hauteur / 2

base_triangle: float = 5
hauteur_triangle: float = 10
aire: float = aire_triangle(base_triangle, hauteur_triangle)
print(aire)
```

En programmation, les noms, c'est comme les blagues :
si tu dois expliquer, c'est qu'ils sont mauvais !

Listes

Listes : motivation

- Je veux manipuler une série de valeurs en mémoire
  - en utilisant un seul nom de variable
  - je ne connais pas forcément à l'avance combien il y en aura

Listes : exemple

- **Variable simple** : une valeur, une « case »

```
v : float = 42.5
v += 2.7
```

- v

42.5

- **Liste** : collection de valeurs (en général, du même type), plusieurs « cases » numérotées

```
# Possibilité 1: prédéfinition
numbers: list[float] = [10.5, 34.6, 0, -12.4, math.pi]

# Possibilité 2: construction dynamique
numbers = []
numbers.append(10.5)
numbers.append(34.6)
numbers.extend([0, -12.4, math.pi])
```

- numbers

0	1	2	3	4
10.5	34.6	0	-12.4	3.14…

Listes : méthodes utiles

- **Ajouter**

```
my_list.append(x)          # Ajoute x à la fin de la liste
my_list.insert(i, x)       # Ajoute x à la position i
my_list.extend([x, y, z]) # Ajoute x, y et z à la fin de la liste
```

- **Supprimer**

```
my_list.remove(x)    # Supprime la première occurrence de x
my_list.clear()      # Vide la liste
x = my_list.pop()    # Supprime et renvoie le dernier élément
```

- **Slicing**

```
my_list[debut:fin]  # Accès à une tranche de la liste
```

ICC-P 03 : plusieurs appels à une fonction

```python
def analyze_string(s: str) -> None:
    print(f"Analyse du string '{s}'")
    print(f"{len(s)} caractères")
    print(f"En majuscules: {s.upper()}")
    print(f"En minuscules: {s.lower()}")
    print("--")


analyze_string("Bonjour")
analyze_string("programmation")
analyze_string("exercice")
```

- pour chaque valeur dans cette liste, assigne-la à la variable `elem` puis exécute le corps de la boucle

```python
for elem in ["Bonjour", "programmation", "exercice"]:
    analyze_string(elem)
```

ICC-P 03 (série) : calculette interactive

```python
if operation == "+":
    result = number1 + number2
    print(f"{number1} + {number2} = {result}")
```

```python
if operation == "+" or "plus":      # erroné
    result = number1 + number2
    print(f"{number1} + {number2} = {result}")
```

```python
if operation == "+" or operation == "plus":
    result = number1 + number2
    print(f"{number1} + {number2} = {result}")
```

Ok, mais long.

🌽? 🇧🇷

```python
if operation in ["+", "plus", "mais"]:
    result = number1 + number2
    print(f"{number1} + {number2} = {result}")
```

Yes ! `x in y` teste si `x` est un élément que `y` contient

Problème : Rechercher dans une liste

- Écrivez une fonction **`find`** qui recherche la première position d'un élément dans une liste, ou retourne `-1` s'il est inexistant

```
odds = [1, 3, 5, 3, 7]

print(find(odds, 3)) # doit retourner 1
print(find(odds, 7)) # doit retourner 4
print(find(odds, 9)) # doit retourner -1
```

Problème : Rechercher dans une liste

```python
def find1(values: list[int], value: int) -> int:
    i = 0
    while i < len(values):
        if values[i] == value:
            return i
        i += 1
    return -1
```

Avec une boucle classique **`while`**

Si la liste à la position **`i`** contient la valeur qu'on cherche, on renvoie **`i`**

Si on sort de la boucle sans avoir fait de **`return i`**, c'est que la liste ne contient pas la valeur cherchée

```python
def find2(values: list[int], value: int) -> int:
    for i, elem in enumerate(values):
        if elem == value:
            return i
    return -1
```

`enumerate()` permet de faire un **`for-in`** à deux variables: la première est l'index de liste, la seconde est la valeur à l'index donné

On utilise le test avec **`in`** et ensuite la méthode fournie **`index()`**

```python
def find3(values: list[int], value: int) -> int:
    if value in values:
        return values.index(value)
    else:
        return -1
```

ICC-T 03 : Tri par fusion

Tri par fusion

entrée : Liste L non triée de nombres entiers, de taille n
sortie : Liste L' triée

Si $n = 1$
    Sortir : L

milieu $\leftarrow \left\lfloor \frac{n}{2} \right\rfloor$
$L_1 \leftarrow$ Tri par fusion(L(1 : milieu), milieu)
$L_2 \leftarrow$ Tri par fusion(L(1+milieu : n), n - milieu)
L' $\leftarrow$ fusion(L_1,L_2)
Sortir : L'

fusion

entrée : Listes ordonnées L_1, L_2 de taille m_1 et m_2 resp.
sortie : Liste L de taille $m_1 + m_2$ également ordonnée

$j_1 \leftarrow 1$
$j_2 \leftarrow 1$
$j \leftarrow 1$

Tant que $j_1 \leq m_1$ et $j_2 \leq m_2$:
    Si $L_1(j_1) \leq L_2(j_2)$:
        $L(j) \leftarrow L_1(j_1)$
        $j_1 \leftarrow j_1 + 1$
    Sinon :
        $L(j) \leftarrow L_2(j_2)$
        $j_2 \leftarrow j_2 + 1$
    $j \leftarrow j + 1$

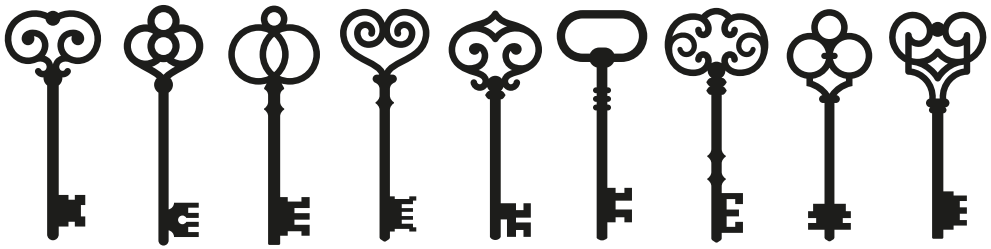
Si $j_1 = m_1 + 1$:
    Tant que $j_2 \leq m_2$:
        $L(j) \leftarrow L_2(j_2)$
        $j_2 \leftarrow j_2 + 1$
        $j \leftarrow j + 1$
Sinon :
    Tant que $j_1 \leq m_1$:
        $L(j) \leftarrow L_1(j_1)$
        $j_1 \leftarrow j_1 + 1$
        $j \leftarrow j + 1$

Sortir : L

Test case : Problème des 100 prisonniers

- 100 prisonniers, chacun numéroté de 0 à 99, et 100 boîtes, également numérotées de 0 à 99.
  Chaque boîte contient une clé différente, correspondant à l'un des prisonniers.
- Chaque prisonnier peut ouvrir 50 boîtes, sans savoir ce que les autres ont fait,
  et doit tout remettre en place.
- Si tous les prisonniers trouvent leur clé, ils sont libérés. Si un seul échoue,
  ils sont tous exécutés.
- La question est donc : **Quelle est la meilleure stratégie et quelle est la probabilité de réussite ?**
  - Approche naïve : $1/2^{100}$
  - Approche optimale : environ 31%
- https://www.youtube.com/watch?v=iSNsgj1OCLA

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99



Simulons ce problème !

Résumé Cours 4 – ICC-P

- Les listes servent à stocker une série de valeurs dans une série de « cases » numérotées depuis 0
- On utilise les crochets, par exemple `[n]`, pour accéder à la nième valeur; on peut faire du slicing avec la notation `[start:end]`
- Le `for-in` marche directement pour itérer sur les listes
- Les listes ont une série de méthodes prédéfinies pratiques

rafael.pires@epfl.ch

Merci