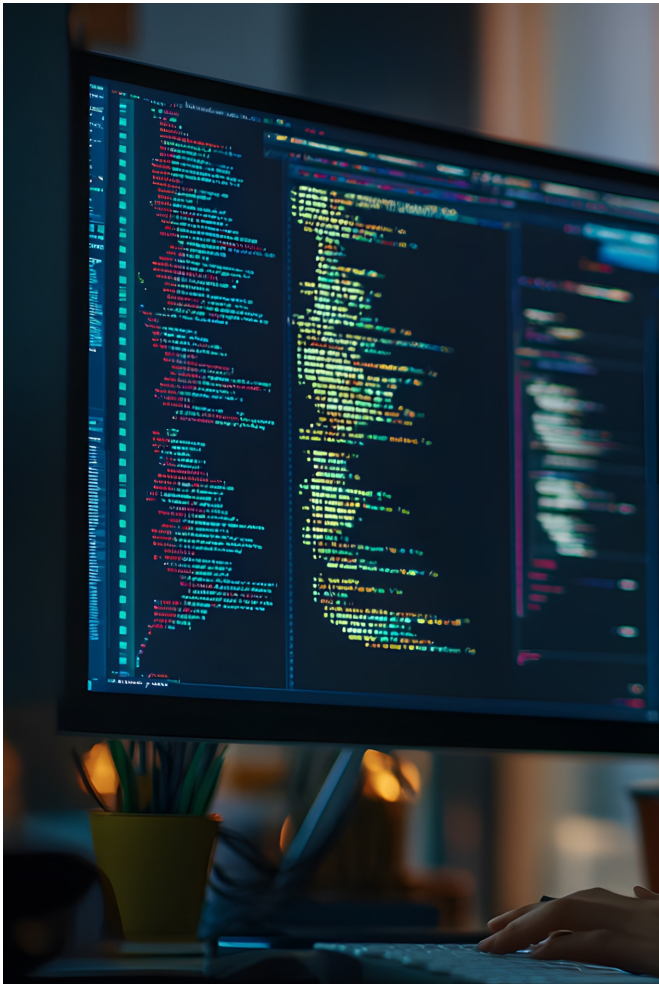


Information, Calcul et Communication

CS-119(k) ICC – Programmation Semaine 2

Rafael Pires
rafael.pires@epfl.ch

Précédemment, dans... ICC-P 01

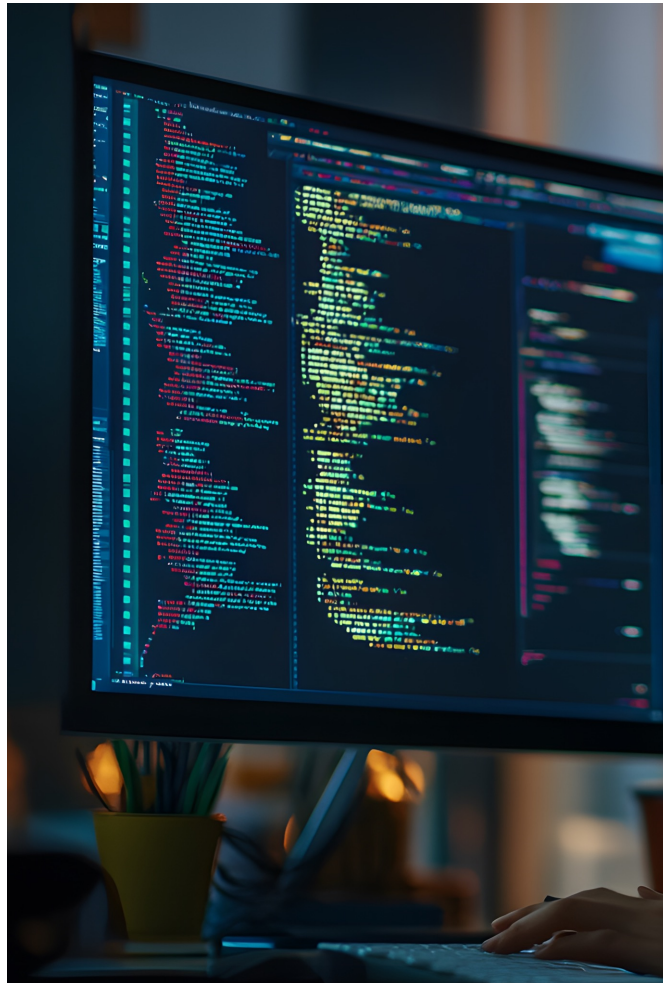


- Nous utilisons **VS Code** pour programmer en **Python**
- Quelques types de base en Python: `int`, `float`, `str`
- **Conversion** entre ces types `print("High " + str(5))`
- Déclaration d'une **variable** avec valeur initiale (*type optionnel*)

```
age: int = 23
height: float = 1.75
my_name: str = "Jean Dupont"
```
- **Méthodes**, **fonctions** et **slicing** pour calculer des valeurs dérivées :

```
upper_name = my_name.upper()
name_length = len(my_name)
my_name = my_name[0:4]
```

Précédemment, dans... ICC-P 01



▪ Méthode

- S'écrit après le nom de la variable suivi d'un point
- Toujours avec parenthèses
- Dépend du type de la variable

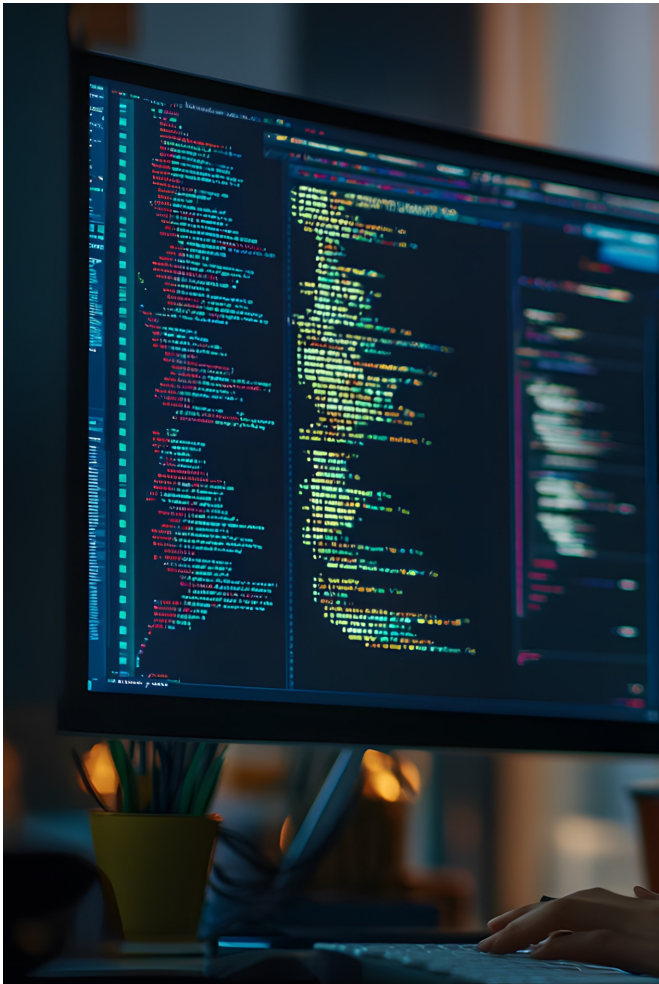
```
upper_name = my_name.upper()
```

▪ Fonction

- S'écrit sans préfixe avec arguments entre parenthèses
- Applicable en général à plusieurs types de données

```
name_length = len(my_name)
```

Précédemment, dans... ICC-P 01



▪ Slicing

- Forme simple: deux indices entre crochet après nom de variable

```
my_name = my_name[0:4]
```

- S'applique aux strings (et *listes*, *tuples* etc.)
- Premier indice optionnel si égal à 0

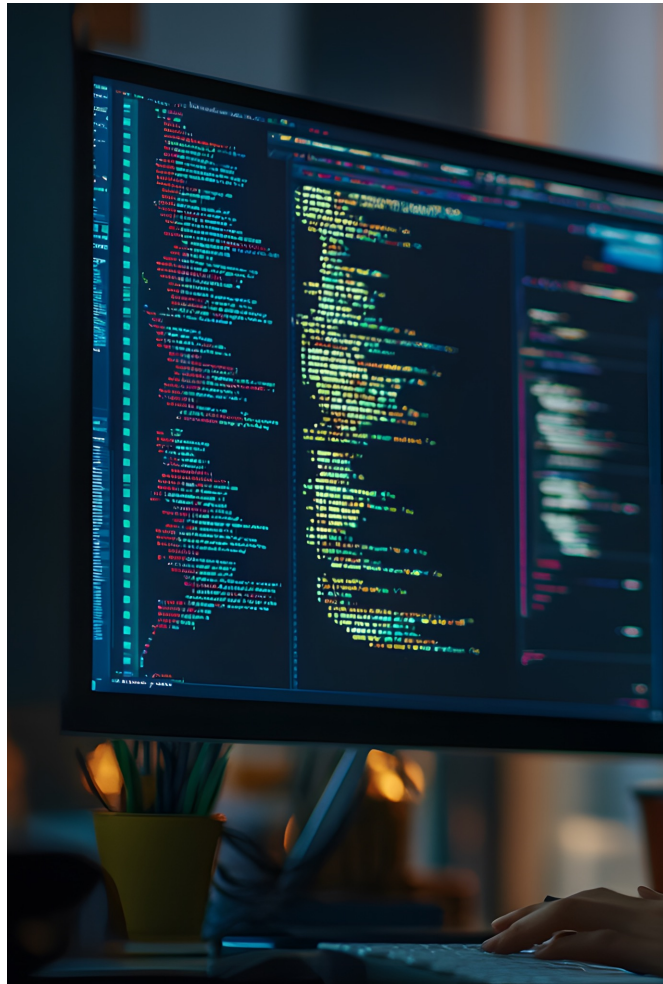
```
my_name = my_name[:4]
```

- Second indice optionnel si égal à `len(variable)`

```
my_lastname = my_name[5:len(my_name)]  
my_lastname = my_name[5:] # équivalent
```

- *Troisième indice possible... On en reparle*

Précédemment, dans... ICC-P 01



- **f-strings**

- Un f-string est un string précédé de **f**

```
print(f"Durée du trajet: {duration} h")
```

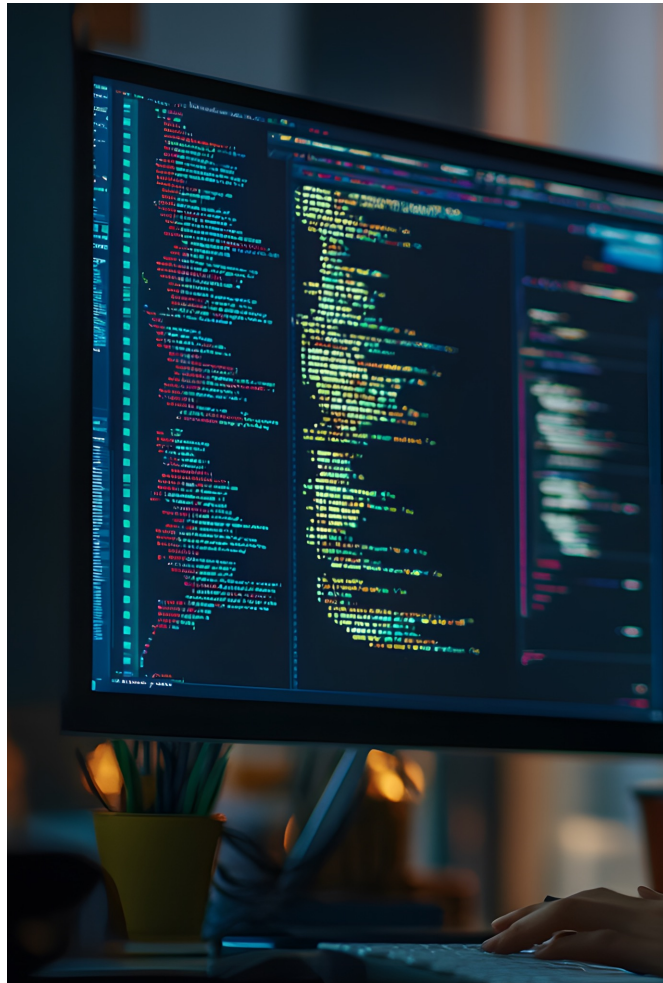
- Les expressions entre accolades **{ }** sont évalués comme code Python et insérées dans le string final

```
print(f"Bonjour, {(1+1) * \"chou\"} !")
```

- Pour insérer une accolade dans un f-string, on la double

```
print(f"On a l'ensemble A = {{x | x > {min_value} }}")
```

Précédemment, dans... ICC-P 01 (exercices)



▪ Division

```
a = 30  
b = 12
```

• Flottante

```
c = a / b # c == 2.5
```

• Entière (*floor division, même si a ou b est un float*)

```
c = a // b # c == 2
```

• Modulo, reste de la division euclidienne

```
c = a % b # c == 6
```

• Une fonction/méthode peut **renvoyer plusieurs valeurs** (*en fait, un tuple*)

```
duration_hours, rest = divmod(distance, speed)
```

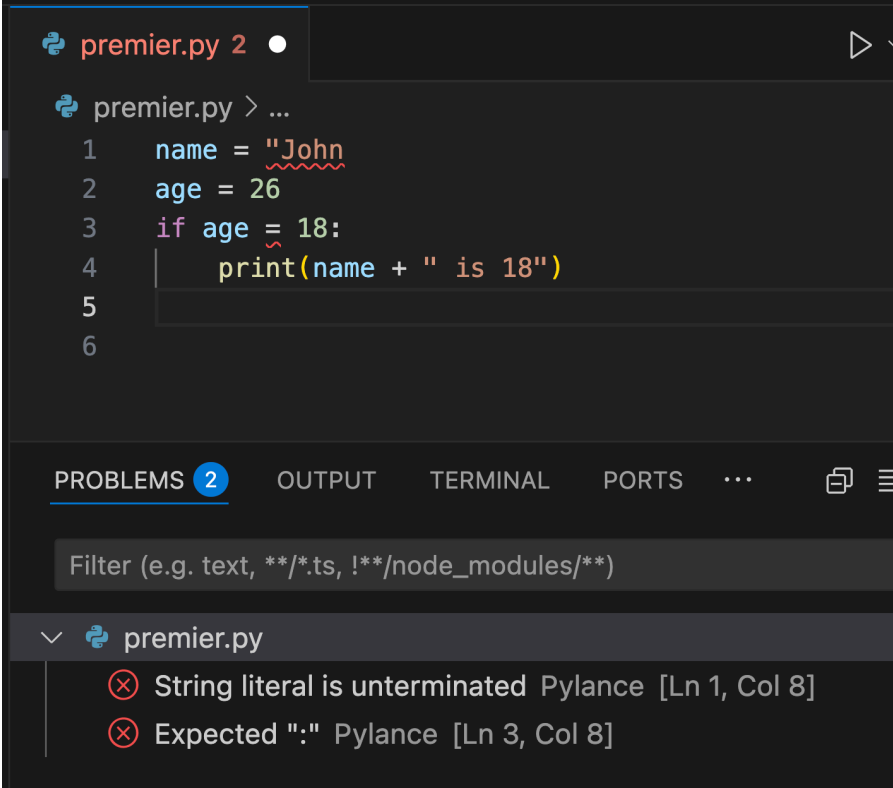
Conseils



Conseil : Erreurs du compilateur/linter

Déplacez votre **curseur** au-dessus de l'erreur pour voir une explication...

... ou affichez l'onglet **Problems** en bas de la fenêtre.



```
premier.py 2
premier.py > ...
1 name = "John
2 age = 26
3 if age = 18:
4     print(name + " is 18")
5
6
```

PROBLEMS 2 OUTPUT TERMINAL PORTS ...

Filter (e.g. text, **/*.ts, !**/node_modules/**)

premier.py

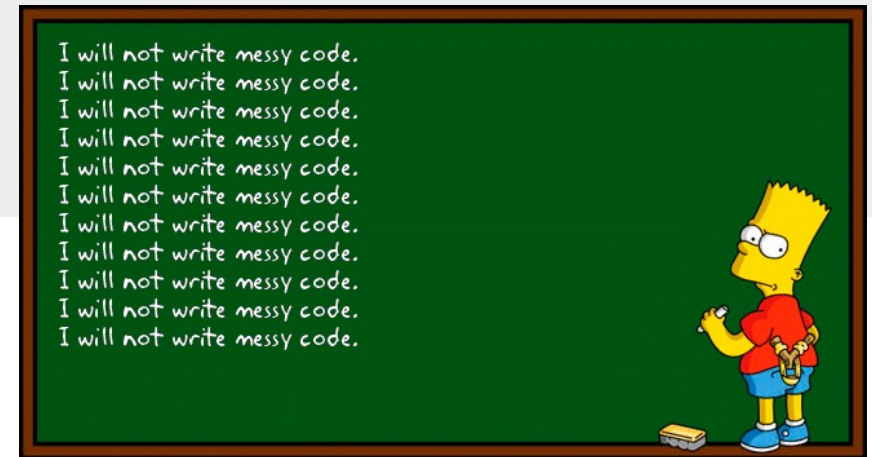
- String literal is unterminated Pylance [Ln 1, Col 8]
- Expected ":" Pylance [Ln 3, Col 8]

Les erreurs sont signalées **en rouge**. Résolvez-les avant de faire tourner votre code.

Conseil : Style du code

- **Nom des variables et méthodes**
 - Pas d'espaces! Pas de caractères spéciaux ou accentués
 - Pas de majuscules
 - Des underscores pour séparer les mots
 - Exemples: `age`, `my_name`, `first_name`

- **Indentation**
 - Nombre d'espaces ou tabs avant le début de la ligne
 - Change la signification du code...



Structures de contrôle

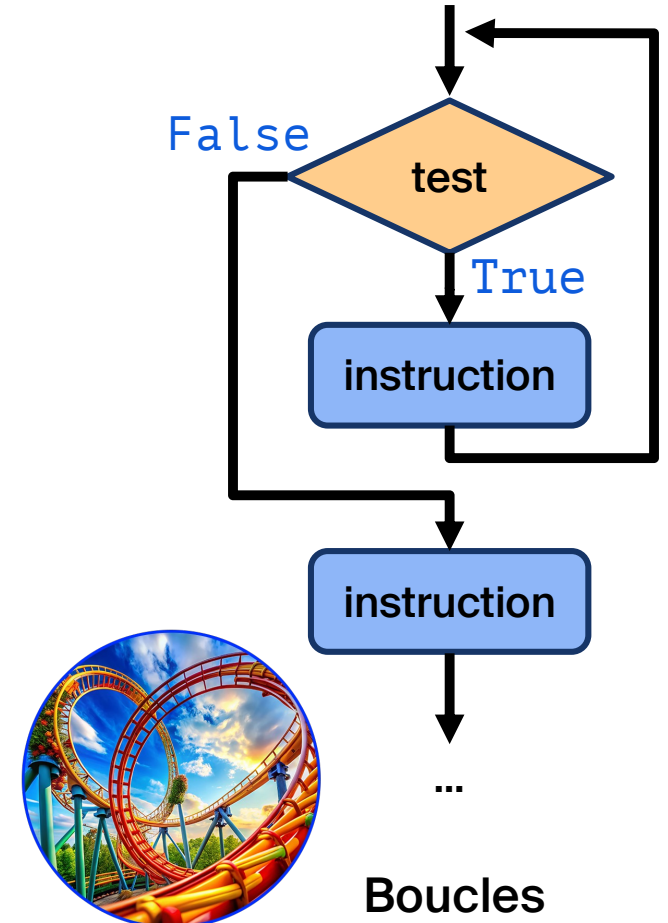
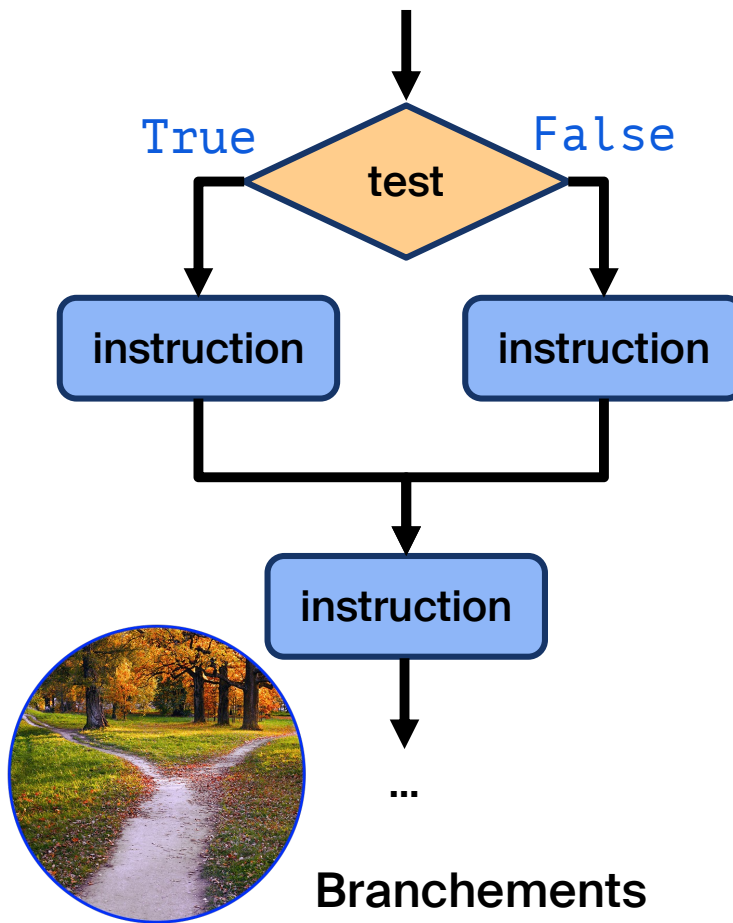
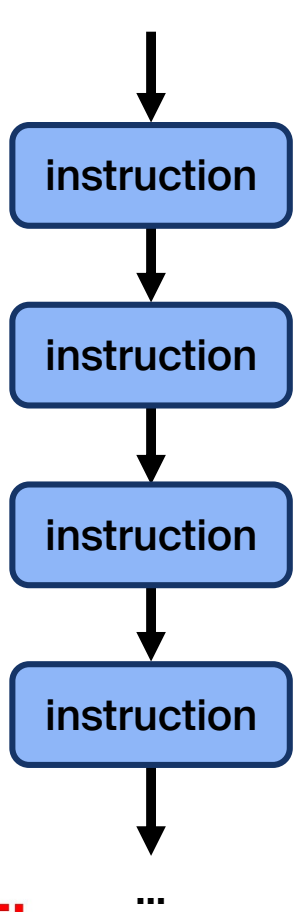


Branchements



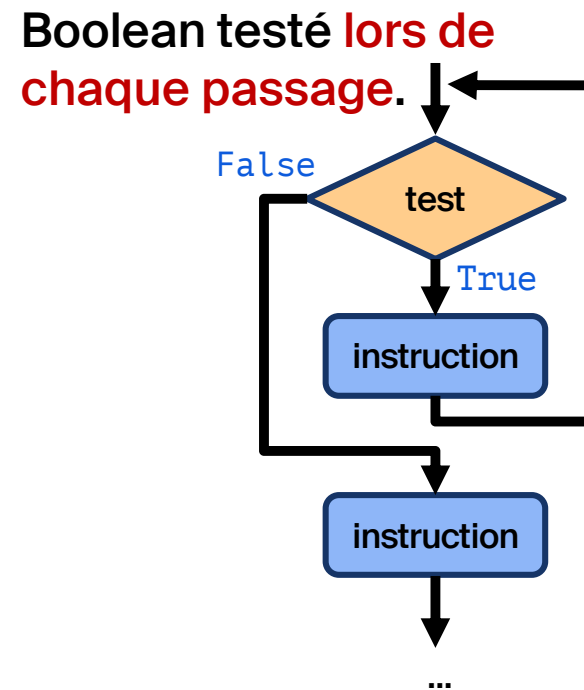
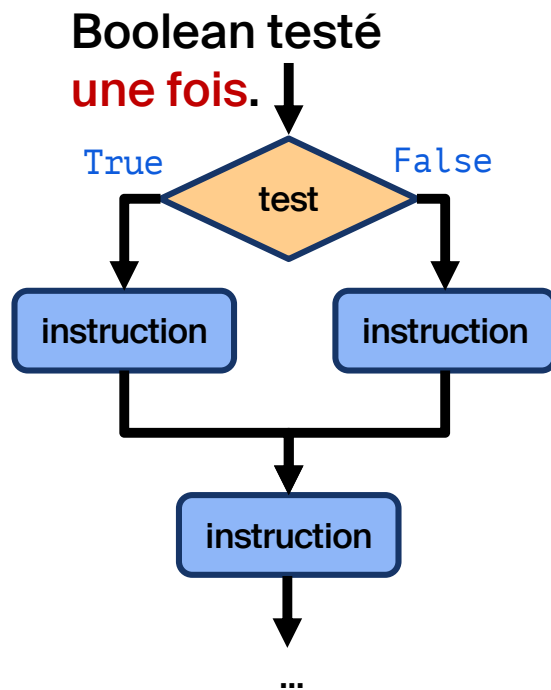
Boucles

Structures de contrôle



Le test

- Chaque test a besoin d'un **bool** pour savoir comment continuer
 - Soit vrai (**True**), soit faux (**False**)



Branchement

```
my_string = "je me demande quelle est la longueur de ceci"
limit = 10

if len(my_string) > limit:
    short_string = my_string[:limit - 1] + "..."
else:
    short_string = my_string

print(short_string)
```

Condition qui produit un bool, suivie d'un deux-points

Que faire si la condition est **fausse** (code *indenté*)

Que faire si la condition est **vraie** (code *indenté*)

Suite normale du programme (code non indenté)

Forme générale d'un if

```
if <bool_condition>:  
    # code if bool_condition is True
```

```
if <bool_condition>:  
    # code if bool_condition is True  
else:  
    # code if bool_condition is False
```

```
if <bool_condition1>:  
    # code if bool_condition1 is True  
elif <bool_condition2>:  
    # code if bool_condition1 is False and  
    # bool_condition2 is True  
else:  
    # code if both conditions are False
```

Comment obtenir des booleans ?

Avec des ints

```
age = 19
if age < 12: # plus petit
if age > 18: # plus grand
if age <= 18: # plus petit ou égal
if age >= 18: # plus grand ou égal
if age == 50: # égal
if age != 13: # non égal
```

Avec des floats

```
price = 1.2
if price < 2.5: # plus petit
if price > 1.3: # plus grand
# égalité et inégalité possibles, mais
# attention aux imprécisions en virgule flottante
```

Comment obtenir des booléens ?

Avec des strings

```
course = "Programmation"
if course == "Programmation":           # égalité
if course.lower() == "programmation":   # sans tester les majuscules
if course.startswith("Prog"):          # test de préfixe
if course.endswith("ation"):           # test de suffixe
if "mmati" in course:                   # test de contenance
```

Avec des booléens

```
x: int = ...

if x > 1 and x < 100: # conjonction: vrai si les deux sont vraies
if x > 1 or x < 100: # disjonction: vrai si l'une des deux est vraie
if not (x > 1):      # inversion: vrai devient faux et inversement
if (not x > 1) or (x > 1 and x < 100): # combinaisons...
```


Boucles



```
i = 7
while i < 100:
    print(i)
    i = i + 7
```

Initialisation

Test

Que faire à chaque itération

Mise à jour de la variable de boucle

Début
Changement
Fin

Pour chaque valeur (appelée *i*) dans ce *range*

```
for i in range(7, 100, 7):
    print(i)
```

Première valeur

Borne supérieure non incluse

Incrément

Que faire à chaque itération

La fonction range()

- **Très utile** pour le **for-in**

- 1 argument : `range(y)` → de **0** (inclus) à **y** (exclu)
- 2 arguments : `range(x, y)` → de **x** (inclus) à **y** (exclu)
- 3 arguments : `range(x, y, s)` → idem, par incréments de **s**

- **Exemples**

```
range(10)           # on part de 0 et on s'arrête avant 10, donc 9
range(0, 10)        # même chose
range(2, 10)        # on commence à 2 plutôt qu'à 0
range(2, 10, 3)     # on incrémente de 3 plutôt que de 1
range(100, 0, -10)  # 100, 90, 80, ..., 20, 10
```

- Dans l'interpréteur : `list(range(...))` Montrera une **liste** avec toutes les valeurs de la range.

ICC-T : Pseudo-code

- Structures de contrôle

- Branchements conditionnels (tests)
- Itérations (boucles)
- Boucles conditionnelles

si $\delta < 0$, alors ...
sinon ...

if, elif, else

pour i allant de 1 à n , (répéter ...)

for-in

tant que $i \leq 10$, (répéter ...)

while

ICC-T : Recherche du minimum dans une liste

$$\begin{cases} L = (13, 47, 18, 15, 11, 19, 46, 18, 15) \\ n = 9 \end{cases}$$

1. On considère le premier nombre de la liste le '**minimum**'
2. On le compare au 2ème nombre, le '**suivant**'
3. Si **suivant** < **minimum**, alors **minimum** ← **suivant**
4. Sinon, on continue, c'est à dire, **suivant** ← celui d'après
5. On revient à l'étape 3 jusqu'à la fin de la liste
6. Le résultat est la valeur de '**minimum**'


Valeur minimale

entrée : liste **L** de nombres entiers, de taille **n**
sortie : la (ou une des) valeur(s) minimale(s)
de la liste

min ← **L**(1)
Pour **i** allant de 2 à **n**
 Si **L**(**i**) < **min**
 min ← **L**(**i**)
Sortir : **min**

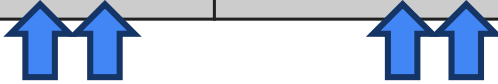
ICC-T : Algorithme d'Euclide

- L'algorithme d'Euclide utilise une boucle conditionnelle pour trouver le plus grand diviseur commun (pgdc) de deux nombres entiers.

pgdc
<i>entrée</i> : a , b , deux nombres entiers positifs <i>sortie</i> : pgdc(a, b)
tant que b ≠ 0 temp ← b b ← a mod b a ← temp Sortir : a 

$$\text{pgdc}(a, b) = \text{pgdc}(a-b, b) = \text{pgdc}(a-k.b, b) = \text{pgdc}(a \bmod b, b)$$

a = 30	b = 12
30 = 2 . 3 . 5	12 = 2 . 2 . 3


pgdc(30, 12) = 6

a	b	temp
30	12	12
12	6	6
6	0	

Debugger

Un programme démarré en mode Debug s'arrête au premier breakpoint trouvé et permet d'avancer pas à pas

État des variables

Outils pour continuer l'exécution

Breakpoint

The screenshot shows a Python IDE in debug mode. The main window displays a Python script named `s02e03.py` with the following code:

```
1 should_continue: bool = True
2
3
4 while should_continue:
5     print("Je vais évaluer un calcul pour vous.")
6
7     number1_string: str = input("Tapez le premier nombre: ")
8     number1: float = float(number1_string)
9
10    number2_string: str = input("Tapez le second nombre: ")
11    number2: float = float(number2_string)
12
13    operation: str = input("Tapez l'opération: ")
14    if operation == "+" or operation == "plus": # avec 'or'
15        result = number1 + number2
16        print(f"{number1} + {number2} = {result}")
17    elif operation in ("-", "moins"): # oh, intéressant
18        result = number1 - number2
19        print(f"{number1} - {number2} = {result}")
20    elif operation == "*":
21        result = number1 * number2
22        print(f"{number1} * {number2} = {result}")
23    elif operation == "/":
24        result = number1 / number2
25        print(f"{number1} / {number2} = {result}")
26    else:
27        print(f"Désolé, je ne connais pas l'opération '{operation}'")
```

The IDE interface includes several panels:

- VARIABLES:** Shows local variables: `number1: 34.0`, `number1_string: '34'`, `number2: 23.0`, `number2_string: '23'`, and `should_continue: True`.
- CALL STACK:** Shows the current execution frame: `<module> s02e03.py (12:1)`.
- BREAKPOINTS:** Shows a breakpoint set at line 12 of `s02e03.py`.
- TERMINAL:** Shows the program's output: `Je vais évaluer un calcul pour vous.`, `Tapez le premier nombre: 34`, and `Tapez le second nombre: 23`.

Red arrows point from the text labels to the corresponding UI elements: "État des variables" points to the Variables panel, "Outils pour continuer l'exécution" points to the Run/Debug toolbar, and "Breakpoint" points to the Breakpoints panel. A red arrow also points to the "Mode" label in the top right corner of the IDE window.

Résumé Cours 2 – ICC-P

- Les **branchements** et les **boucles** sont des structures de contrôles essentielles en programmation
- Les booléens sont à la base des décisions qu'on prend avec **if** ou **while**
- On obtient ces booléens avec, entre autres
 - des **comparaisons** (p. ex. sur des types numériques)
 - en appelant des **méthodes** (p. ex. sur un string)
- Les booléens peuvent se **combiner logiquement** entre eux avec les opérateurs **and**, **or** et **not**
- La fonction **range()** permet de faire des boucles plus simplement avec **for-in**

rafael.pires@epfl.ch



EPFL

Merci