Python 11 – Instructions conditionnelles

I – Expressions booléennes

Une expression booléenne est une expression qui est évaluée à **True** ou **False** par Python.

Les opérateurs classiques à connaître :

```
>>> annee = 2025
>>> annee == 2025     Opérateur d'EGALITE
True

>>> annee != 2025     Opérateur de DIFFERENCE
False

>>> annee >= 2025     Un opérateur de COMPARAISON
True

>>> annee < 2025
False

>>> "Bo" in 'bonjour'  Opérateur d'APPARTENANCE
False
```

II – Structure IF ELIF ELSE

Structure générale : explications

L'instruction conditionnelle peut se décomposer en plusieurs blocs mais ATTENTION : **seul l'un des blocs sera activé** : celui donc la condition est évaluée à True en premier**. Ensuite, on quitte définitivement** la structure (même si les autres expressions pourraient être évaluées à True).

Un mot-clé obligatoire : **`if + CONDITION`** pour commencer. Signifie SI.
Des mots-clé optionnels : **`elif + CONDITION.`** Signifie SINON SI...
Un mot-clé final optionnel : **`else`** (sans condition) : Signifie SINON (« si aucun autre cas n'a été validé »)

Voir les exemples ci-dessous.

Deux façons de gérer des fonctions contenant un SI

Version propre avec une seule "porte de sortie"

```
1   def appreciation(note):
2       if note > 13:
3           app = "Bien"
4       elif note < 9:
5           app = "Insuffisant"
6       else:
7           app = "Moyen"
8       return app
9
10  a = appreciation(4)
```

AVANTAGE : Un seul return, facile à surveiller

DESAVANTAGE : oblige le lecteur à tout lire.

Déroulé : L1(déclaration) – L10(appel) – L1 – L2(False) – L4(True) – L5 – L8 - L10.

Version avec plusieurs "portes de sortie"

```
1   def appreciation(note):
2       if note > 13:
3           return "Bien"
4       elif note < 9:
5           return "Insuffisant"
6       else:
7           return "Moyen"
8
9   a = appreciation(4)
```

AVANTAGE : Le lecteur voit plus rapidement la réponse qu'il cherche.
DESAVANTAGE : Plusieurs sorties à surveiller si on veut vérifier automatiquement que la fonction répond correctement.

Déroulé : L1(déclaration) – L9(appel) – L1 – L2(False) - L4(True)-L5-L10.

III – Prédicat

C'est **une fonction qui renvoie un booléen**.

On parle de prédicat puisque la fonction prédit une propriété à partir des paramètres reçus.

Créer un prédicat ne requiert pas nécessairement d'utiliser un **`if`**, on peut parfois juste fournir l'expression qu'on veut évaluer derrière votre **`return`**.

Exemple : la note de l'étudiant permet-elle de valider un semestre (vaut-elle au moins 10) ?

```
1  def est_valide(note:int) -> bool:
2      '''Prédicat, True si note suffisante'''
3      return note >= 10
```

```
1  def est_valide(note:int) -> bool:
2
3      '''Prédicat, True si note suffisante'''
4      if note >= 10:
5          return True
6      else:
7          return False
```

IV – Prédicat en tant qu'expression booléenne

CAS 1 Imaginons qu'on dispose d'un prédicat **contient_le_mot()** qui permet de tester si une chaîne de caractères contient un mot qu'on recherche.

```
def contient(chaine:str, mot:str) -> bool:
    return mot.lower() in chaine.lower()
```

```
>>> contient("Je veux me plaindre de mes notes bla bla", "Plaindre")
True
```

Bonne pratique de programmation

```
if contient(c, "montre"):
    instruction1
    instruction2
...
```

Mauvaise pratique de programmation

```
if contient(c, "montre") == True:
    instruction1
    instruction2
...
```

CAS 2 : Imaginons qu'on veuille agir lorsqu'un mot n'est pas présent dans un message. On dispose d'un prédicat **contient()** qui permet de savoir si un mot est contenu dans une chaîne de caractères.

Très mauvaise pratique de programmation

```
if contient_le_mot(c, "montre"):
    pass
else:
    instruction1
    instruction2
...
```

Assez mauvaise pratique de programmation

```
if contient(c, "montre") == False:
    instruction1
    instruction2
...
```

Bonne pratique de programmation : utiliser le mot-clé **not** : ce mot-clé inverse le booléen situé derrière lui.

```
>>> condition = False
>>> condition
False

>>> not condition
True
```

On peut alors écrire plus proprement :

```
if not contient(c, "montre"):
    instruction1
    instruction2
...
```

Le ET logique avec Python

On rappelle que **a and b** n'est vrai que lorsque a et b sont vrais.

Le **and** permet de remplacer deux if imbriqués l'un dans l'autre. Exemple : on veut agir lorsqu'on tombe sur des montres de luxe :

```
if contient(c, "montre"):
    if contient(c, "luxe"):
        instruction1
        instruction2
```

Ou plus simplement :

```
if contient(c, "montre") and contient(c, "luxe"):
        instruction1
        instruction2
```

Le OU logique avec Python

On rappelle que **a or b** n'est faux que lorsque a et b sont faux.

Le **or** permet de remplacer un if suivi d'un elif. Exemple : on veut agir sur les montres ou les objets de luxe :

```
if contient(c, "montre"):
    instruction1
    instruction2
elif contient(c, "luxe"):
    instruction1
    instruction2
```

Ou plus simplement :

```
if contient(c, "montre") or contient(c, "luxe"):
        instruction1
        instruction2
```

V – Deux remarques finales

Remarque 1 Python évalue à False les contenus vides (0, None, []...). On peut donc agit si un tableau est non vide ou si il est vide de cette façon

```
if tableau:                 if not tableau:
    # NON VIDE                  # VIDE
    instructions                instructions
```

Remarque 2 Une erreur typique est de ne pas se souvenir que Python lit un booléen de chaque côté du and ou du or.

`"M" or "C" in texte` *veut dire en réalité :*

Si "M" (donc True!) ou si "C" apparaît dans texte.

Cette condition est donc toujours vraie ! La vraie condition à taper :

```
"M" in texte or "C" in texte
```